2018-09

A BLOCKCHAIN TESTBED FOR DOD APPLICATIONS

Shaw, Markus R.

Monterey, CA; Naval Postgraduate School

http://hdl.handle.net/10945/60460

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A BLOCKCHAIN TESTBED FOR DoD APPLICATIONS

by

Markus R. Shaw

September 2018

Thesis Advisor:	Cynthia E. Irvine
Second Reader:	Gurminder Singh

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE A BLOCKCHAIN TESTBED FOR DoD APPLICATIONS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Markus R. Shaw	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Although initially introduced to support the Bitcoin cryptocurrency, many blockchain technology applications have been envisioned. As a result, blockchain platforms have been created and others are under development. The Department of Defense (DoD) needs blockchain platform testbeds so that applications beyond cryptocurrency can be explored. To solve this problem, we decided to construct a small blockchain testbed. We developed criteria to aid in selecting a blockchain platform for use in a testbed. Then, we evaluated a set of blockchain platforms against our criteria, selected the highest-ranking platform, and used it to create the testbed. We installed and exercised applications provided by the platform's developers to test its behavior. Study of the services provided by the platform allowed us to propose the design of an application that could be tailored for DoD use. The selected blockchain platform, Hyperledger Fabric, was hosted in lightweight virtual machines called Docker containers and can be used for design and experimentation on applications and blockchain networks. This lowered the effort and resources required to configure and set up blockchain networks. Hyperledger Fabric is an example of a blockchain platform that can support more use cases beyond cryptocurrency.

14. SUBJECT TERMS blockchain, blockchain network, test bed, virtual machine, Docker, blockchain selection criteria, Hyperledger Fabric	**15. NUMBER OF PAGES** 111
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

A BLOCKCHAIN TESTBED FOR DOD APPLICATIONS

Markus R. Shaw
Civilian, Scholarship for Service
BS, California State University: Monterey Bay, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2018

Approved by: Cynthia E. Irvine
Advisor

Gurminder Singh
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Although initially introduced to support the Bitcoin cryptocurrency, many blockchain technology applications have been envisioned. As a result, blockchain platforms have been created and others are under development. The Department of Defense (DoD) needs blockchain platform testbeds so that applications beyond cryptocurrency can be explored. To solve this problem, we decided to construct a small blockchain testbed. We developed criteria to aid in selecting a blockchain platform for use in a testbed. Then, we evaluated a set of blockchain platforms against our criteria, selected the highest-ranking platform, and used it to create the testbed. We installed and exercised applications provided by the platform's developers to test its behavior. Study of the services provided by the platform allowed us to propose the design of an application that could be tailored for DoD use. The selected blockchain platform, Hyperledger Fabric, was hosted in lightweight virtual machines called Docker containers and can be used for design and experimentation on applications and blockchain networks. This lowered the effort and resources required to configure and set up blockchain networks. Hyperledger Fabric is an example of a blockchain platform that can support more use cases beyond cryptocurrency.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ABAC	attribute-based access control
API	application programming interface
BP	blockchain platform
CA	certificate authority
CSS	cascading style sheets
DAG	directed acyclic graph
DoD	Department of Defense
FAQ	frequently asked questions
GNU	GNU’s not Unix
GPL	general public license
HLF	Hyperledger Fabric
HTTP	hypertext transfer protocol
JSON	JavaScript object notation
MIT	Massachusetts Institute of Technology
MSP	membership service provider
OS	operating system
PBFT	practical Byzantine fault tolerance
PKI	public key infrastructure
PoS	proof of stake
PoW	proof of work
SDK	software development kit
TIOBE	the importance of being honest
URL	uniform resource locator
YAML	YAML ain’t markup language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Cynthia Irvine, and my second reader, Dr. Gurminder Singh, for helping me through the process of writing my first academic thesis. I also want to thank LCDR Scott Tollefson for being someone who I could bounce ideas off and talk to about technical aspects of Hyperledger Fabric.

I also thank the friends I have made at NPS, both Scholarship for Service students and active duty military students, who helped me feel welcome during my time here. I thank my parents, David and Gloria Shaw, for always believing in me and encouraging me to follow my dreams and aspirations. I am grateful and lucky to have a loving family that supports my academic and professional career aspirations. Last, but not least, I would like to thank Naval Support Activity Monterey Police Officer Eddie Macias for his professionalism, positive attitude, and daily smile.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Following the introduction of Bitcoin in 2008 [1], there has been considerable interest in using the blockchain technology underlying Bitcoin for other purposes. Many blockchain platforms now exist, and others are being developed. Cryptocurrency or token based transactional systems are the main use cases for blockchain platforms. The technology presents potential in the area of distributed application platforms. This study seeks to explore a small portion of that potential with a blockchain platform that is sufficiently versatile to allow a variety of use cases. The United States government and military could use applications that incorporate blockchain technologies in the future. Cryptocurrency and token-based applications are not likely to be pursued by this sector. It is important to research blockchain platforms (BPs) so that applications beyond cryptocurrency can be explored for use in the government and military. Testbeds for experimenting with blockchain technology are needed.

This study defined criteria to evaluate blockchain platforms (BPs), then surveyed multiple BPs. One BP, Hyperledger Fabric (HLF) was selected and applications were tested. An application design using existing HLF platform behavior was proposed and discussed. The programmable nature of HLF smart contracts is an important feature that provides freedom to develop custom application solutions with this BP.

First, in Chapter II, we give background on blockchain platforms including key components, history of academic development, the power usage concerns for certain blockchain technologies, and sample use cases. Next in Chapter III, we define selection criteria and metrics that we will use to choose a BP for our testbed. After that, in Chapter IV, we evaluate a group of blockchain platforms using the selection criteria and metric system to measure their level of appropriateness for the testbed. Some of these metrics are subjective in nature, but they aid in selecting a blockchain platform that can support a diverse set of use cases instead of just cryptocurrency or token exchange. The selection process will be discussed, providing discussion of Hyperledger Fabric (HLF), the selected platform, and the runner up platforms. A matrix of the selection criteria showing how the candidate blockchain platforms were ranked is provided in Chapter IV.

In Chapter V, Hyperledger Fabric’s architecture including the utilities that operators use to manage and develop applications for the platform will be discussed. In Chapter VI, sample applications of HLF provided by its developers will be discussed. The capabilities of and behavior covered by these sample applications will be the focus of the discussion. Also in Chapter VI, we will propose a design for an application that uses the capabilities provided by the HLF platform and could be useful in the context of short-term, limited operation. While the application is not implemented as part of this work, a discussion of future work and aspects of the design that can lead to a working implementation will be provided in Chapter VII. Also, Chapter VII will contain the lessons learned from reviewing different blockchain platforms and investigating the HLF platform.

This chapter gave a brief outline of what this study will attempt to accomplish and gives a preview of the subsequent chapters. The next chapter will provide background on the components of blockchain technology, the academic history of those components, energy consumption concerns associated with blockchain technology, and recent use cases of blockchain technology.

II. BACKGROUND

A. BLOCKCHAIN OVERVIEW

Blockchain is the foundational technology for crypto currency applications. The data structure is, in effect, a ledger that contains data about information in use in an application or system [2]. A blockchain is an application that executes on machines that could be different platforms but are still compatible with the same application rules and protocols. Blockchain applications could be considered a platform themselves and will be referred to as blockchain platforms (BPs) for the rest of the chapter. The style of application that blockchain follows is as a distributed system which is a paradigm that allows different nodes, or computers in this case, to work together and coordinate to achieve a common outcome, according to Bashir [3]. This means that separate devices work together on a service, or problem within the blockchain application over the internet, or a network.

The next section will include a discussion of decentralized applications, a paradigm used in blockchain applications.

1. Decentralized Applications in General

Historically, the convention for information systems has usually been centralization as opposed to decentralization. Returning to Bashir's book, centralized systems contain a central point of authority and/or trust on the network which could be a central server, which follows the client—server model, or a system administrator for that system [3]. Siraj Rival, in the book *Decentralized Applications* [4], states that a centralized application network setup will have one specific node, a master node, that controls the operations of all other nodes on the network, but a decentralized network does not have a controlling node that the other nodes rely upon for correct operation. A failure of the master node on a centralized system would mean the system cannot operate. In contrast a failure of one node on a decentralized system will mean that the application network can still operate without it [4]. The computing paradigm followed by blockchain applications involves a decentralized network of nodes.

While these two paradigms are different, they both are capable of being distributed in nature.

2. Distributed Applications in the Context of Decentralized Applications

Centralized and decentralized applications can both be distributed. Distributed computing means that computation for the system or application is spread across the entire network of nodes to accomplish some task [4]. Depending on the implementation, blockchain technology is both decentralized and distributed. The cryptocurrency Bitcoin is distributed because of the timestamping on the ledger at the core of the Bitcoin network [4]. This is one example of a blockchain technology that is both distributed and decentralized.

3. The Genesis Block

The genesis block is the first entry in the chain for any blockchain [5]. The genesis block structure is the foundation upon which all the interactions with the blockchain [5]. This would be the blockchain's 0th block on the chain with no backwards pointers and would contain no transactions since none have been created by users yet [5].

4. Consensus Mechanisms

Blockchain applications are distributed around multiple nodes or computers [6]. Since the ledger is shared between them there needs to exist a way to keep those nodes on the same page. Consensus is the mechanism that validates the writing of information to the ledger for any given BP [6]. Multiple implementations of consensus exist and each of them contributes different attributes to the BP. A couple of consensus mechanisms are proof of stake (PoS) and proof of work (PoW) [6].

a. Proof of Work

PoW is used by the Bitcoin network. Transactions are placed into a memory pool (or mempool); the first miner to solve a computational problem associated with a transaction is rewarded with the new bitcoin and his or her block is written to the ledger [6]. The bitcoin that the miner is rewarded with is the currency that is traded on the Bitcoin platform. These computational problems are hard to solve, through a process called mining, which is the reason that this mechanism can promote security, at the cost of significant and increased energy consumption required to solve the problem [6]. The computation power required to create a block is hard to replicate, because it requires brute force hashing, but the result of the computation is easy to verify on the network [6]. Since the problems are hard to solve, it is not worth the time and effort for a malicious actor to repeat the brute force computation to try and cheat in the network.

A known attack on this approach to consensus, however, is the "51% attack." This attack works when a malicious actor or actors' control over 51% of the computation power in the Bitcoin network [7]. When over half of the "hashrate" of a BP using proof of work is controlled by one party then the platform can be manipulated and controlled by that party [7]. This means that the ledger could be altered, and legitimate transactions could be reversed, or the ledger could be rewritten, and illegitimate transactions could be added to the ledger [7]. Power consumption can be a problem with BPs and will be discussed below in the ***Power Consumption of Blockchain*** section of this chapter.

b. Proof of Stake

The PoS model the algorithm attributes a new block to participants deterministically, without requiring a high level of computation power [6]. A participant stakes a claim to coins, or currency in this model, and the participant with a greater claimed stake has a better chance at receiving, therefore becoming the validator of, the new block, and subsequently the receive currency associated with the platform. [6]. This means that participants do not have to spend much computation power or energy on mining for blocks or coins, they stake a claim and "are rewarded through transaction fees" generated when a block is validated, and the cryptocurrency is transferred in a transaction [6]. Additionally, the coins or currency that participants stake a claim to are created and exist in the network at the start; they are not mined by any entity on the network [6]. Participants are not competing to mine or create new blocks or coins in this model, but rather they are competing to gain access to them.

c. Zero-Knowledge Proof

The zero-knowledge proof model is an older idea that could be applied to consensus within BPs. First suggested by Goldwasser, Micali, and Rackoff in 1985 at Massachusetts Institute of Technology (MIT), zero-knowledge proofs allow for one party to prove truth without revealing any information about how this truth was known to a verifying party [8]. This concept can work with the public key cryptography, but here we give an analogue example. Lukas Schor gives the example of an individual trying to prove to a blind person that the blind person is holding two balls that are colored differently; the method of proof includes the blind person hiding the balls and then showing them one at a time to that individual for verification, and through this process it can be proved that the two balls have different colors [8]. This process is done repetitively because the odds of a false positive decrease the more times this exercise is done [8]. There are some properties that a zero-knowledge proof must have to be considered valid: completeness, soundness, and privacy (or zero-knowledge) [8]. Completeness means that a correct input generates a correct input, soundness achieved is when the input cannot make a false positive occur in the output, and the privacy or zero-knowledge aspect means that input does not end up in the hands of anyone but verifier and the only thing that is made public is whether the output is true or false [3], [8]. These properties must be considered when incorporating zero-knowledge proof functionality into an application

How this relates to BPs is dependent on a concept that builds on the zero-knowledge proof theory. Zk-Snarks is a term introduced by Bitansky, Canetti, Chiesa, and Tromer in 2012 to define an implementation variation of zero-knowledge proofs intended to work on BPs [8]. This implementation adapts zero-knowledge proofs for the environment of a BP. The concept of zk-snarks is optimized in BPs to make sure that the fact that a valid transaction has occurred is the only thing other entities on the network are aware of [8].

d. Practical Byzantine Fault Tolerance

This consensus method is a work that builds upon Byzantine-fault-tolerant algorithms. Byzantine-fault tolerant systems deal with the “Byzantine Generals” problem described by Lamport, Shostak and Pease in 1982 [9], [10]. Description of this problem deals in a distributed system uses the metaphor of Byzantine generals during a wartime operation. The generals represent different nodes in a network, in this case a blockchain network, and the problem is how to get the generals, or nodes, to reach consensus on a piece of information without having bad actors, i.e., traitorous Byzantine generals, influence the decision making [9]. The practical Byzantine-fault tolerance (PBFT) approach tries to present a solution to this problem. The algorithm presented by Castro and Liskov replicates a state machine across distributed nodes which "maintains service state and implements service operations" [11].

B. FORKING

Forking in a blockchain is like forking on a version control repository. Before explaining forking in a blockchain, it would be appropriate to explain version control software and a special data structure called a Directed Acyclic Graph (DAG). Version control software is used in collaborative development environments to save file history, track changes made to files, and track who made changes to files in a repository [12]. This type of functionality is especially useful for introducing small changes to a project with the ability to roll back to an earlier point in the project's history should issues arise.

One way to keep track history in a repository is in a straight line as represented in the Figure 1 in chapter four of Eric Sink's electronic book on version control software [13].

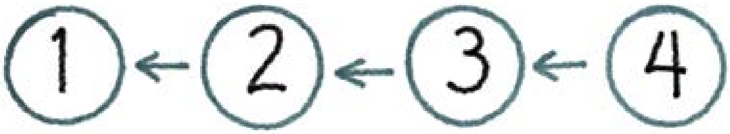


Figure 1. History represented as a line. Source: [13].

In this case versions of a repository are tracked sequentially, one after the other in order as changes are added as pictured in Figure 1. This linear model of version tracking only allows building on the latest version of the project. This can cause issues if two people are working on changes, and one person commits their changes before the second person;

the second person would not be able to commit their changes in this model because they are no longer based on the most recent version [13].

The DAG version of history tracking allows for a non-linear progression of a project in a version-controlled repository [13]. Figure 2 is an example of a DAG representing version tracking that has a non-linear history.

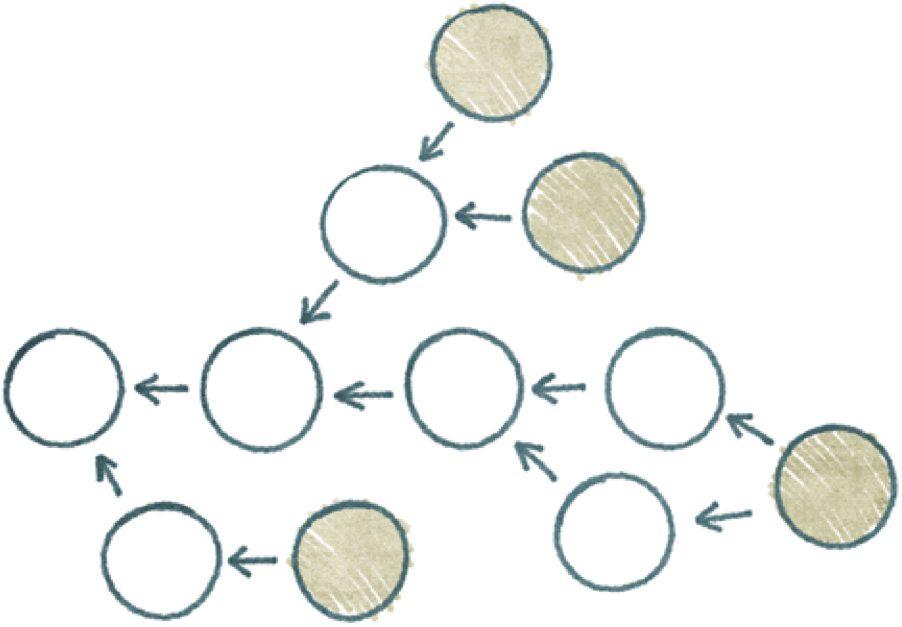


Figure 2. A DAG representation of version tracking. Source: [13].

Collaborators are not limited to working on the most recent version and need not worry about their changes not being accepted because the repository has unexpectedly changed before making a commit [13]. In the DAG model of history tracking for a repository, new versions of a project can use any point in the project's history as a launching point for a new version [13]. An example of this situation is given in chapter four of Sink's book where a change is made to a project repository that contains nodes that represent points in time of the project's history [13]. A programmer could use any of the nodes as a basis for updating the project instead of the last one in chronological order according to Sink's description of a DAG. This would cause the repository to fork off into a new branch of development which could merge back into the main line at any point or continue indefinitely [13].

Sometimes developers on a BP, or a group not related to the blockchain, can start a new project using existing project code as a starting point. Two types of forks exist for blockchains: hard-forks, and soft-forks [14]. A hard fork means that the BP's ledger or chain branches off to form a new chain where the blocks follow different rules than the original chain and they will never converge in the future [14]. This is the DAG concept in action, except that the branch is permanent and can result in a completely new BP that was based on another one. A soft fork is like a hard fork, but instead of one chain always running parallel to another chain and never converging, this new chain supports both the original chain's rules and the new chain's rules [14]; this is treated as an update and is not mandatory for all nodes on this blockchain network. The soft-fork is only then used by the nodes that update to use the forked software [14]. If the nodes that run the software drop off the network, or no nodes adopt the software then the soft-fork will cease to exist on the blockchain network, meaning the soft fork will end, and the chain that it branched from will continue without that fork [14]. In a hard fork, when the platform branches out it is cloned, meaning in a crypto-currency use case that users on the original chain will carry over the amount of crypto-currency they had to the new chain [14]. This would be the initial values that users have on the new chain but going forward the two chains would not be connected, and the transactions would be different between the two [14].

C. SMART CONTRACTS

Transactions that occur in blockchain networks have roots in the concept of smart contracts. The idea of a smart contract first appeared in the 1990s; the concept itself was implemented in Bitcoin with limited functionality [3]. The smart contracts in Bitcoin are the only used to transfer units of the Bitcoin cryptocurrency between users of the software [3]. In other blockchain implementations, smart contracts can become more. The actual definition of a smart contract is "a secure and unstoppable" program that is "executable and enforceable" [3]. The logic programmed into a smart contract is executed on the blockchain; these transactions can be customized to meet the requirements of an application. Smart contracts act like classes in C++ or Java, their code contains variables and functions to execute the transaction logic [15].

D. TYPES OF BLOCKCHAIN

BPs have a few attributes that govern access to the ledger: permissioned, non-permissioned, private, and public. A public blockchain is one where anybody can contribute, and nobody is trusted an example of this is Bitcoin [3]. A private blockchain is only available to members of a group, and the ledger that records transactions on the network is only available to that group [3]. Both private and public blockchains can be permissioned if access controls are put into place [3]. This makes sense in a blockchain where the users are not anonymous.

E. HISTORY OF THE COMPONENTS IN A BLOCKCHAIN

This section will go over the history of the components of blockchain technology which were first widely used in the Bitcoin platform. Most of the science associated with Bitcoin was presented in the years prior to its launch. In Bitcoin, the data structure that models the ledger was borrowed from a concept that was introduced between 1990 and 1997 by Habar and Stornetta who wrote about using timestamping to make a digital notary equivalent [16].

A major data structure in used by Bitcoin is called a Merkle tree. This structure is a binary tree that is built from the pairs of nodes up to the root node according to Antonopoulos [17]. Each leaf node is a transaction, and parent nodes are created by hashing transaction data in pairs until the final node, or Merkle root is computed and becomes the digest for all the transactions in a block on the ledger [17]. If an odd number of transactions exist at the point the tree is constructed, the last transaction is duplicated [17]. Once the hash for the Merkle root has been computed it can be used to verify all of the transactions in a block even if the ledger was downloaded from and untrusted source [16], [17].

The early version of PoW includes a message transmitter that computes a function that is moderately expensive for deterring illegitimate transmissions like spam email [18]. This problem is related to the Sybil attack, in which a network is overwhelmed by sockpuppet nodes, or multiple nodes that are controlled by an adversary, and the integrity of consensus in the network is compromised [16].

F. HOW HAS BLOCKCHAIN TECHNOLOGY BEEN USED?

This section is a sample of the use-cases and applications that BPs have been used for. These are among the first applications that used a blockchain.

1. Cryptocurrency

In the case of Bitcoin, a cryptocurrency application works by use of public key cryptography allowing users to send and receive units of the cryptocurrency to other users with no third-party intervention [2]. In public key cryptography an identity is tied to a public key that the public knows and has access to, and a private key that is mathematically linked to the public key for identity verification but kept secret from the public [19].

2. Financial Transactions

The transactional nature of blockchain has been applied to financial applications. Between 2015 and 2016 financial institutions experimented with blockchain for working with bonds, fixed income trading, and other facets of the financial sector [20].

G. HOW COULD BLOCKCHAIN TECHNOLOGY BE USED?

This section will discuss an application that could thrive in the blockchain application environment: chain of custody for law enforcement or item delivery. The definition of chain of custody from Duke Law is that all information for a file that travels from one destination to another must be detailed and accounted for [21]. This could be applied to the delivery of physical or digital items. This idea could be associated with internal inventory transit, or for moving items and supplies around the country.

H. POWER CONSUMPTION OF BLOCKCHAIN

This section provides additional detail about blockchain power consumption, using Bitcoin as an example. Before power usage is addressed, the concept of hashing will need to be discussed, as it is a big part of the computational problems Bitcoin miners must solve.

1. Hashing Explained

A hashing algorithm takes data, operates on it, and produces an output that can be used for integrity in computer systems. Specifically, a hashing algorithm takes variable length values of text or contents of files and outputs a cryptographic sum or digest that is fixed in length [22]. This process is supposed to work on any data that is represented in a computer. The hashing process is different from the encryption and decryption process in a very fundamental way. Hashing is not meant to preserve information; information could be lost as the hashing algorithm converts the input to a fixed length value in a process that is not reversible like the encryption process is [22]. Hashing supports integrity of data but does not address confidentiality or availability of data. The mathematics behind hashing provides the ability to tell when a file or piece of text has changed because a hash sum or digest will, with reasonable assurance, be unique to the file or text that was used to produce it [22]. Hashing is used in Bitcoin's consensus mechanism and is the major factor contributing to the substantial power usage of a machine running Bitcoin's consensus algorithm [1].

2. Power Usage in Bitcoin Mining

Mining is executed in a brute force manner, meaning that many values, numbers that are only used once and called nonces, must be put through a hash function before a match is found and this is where the most energy is consumed [23]. Bitcoin mining is a competition, and winning that competition means finding a hash that matches the in leading zeroes [6], [23]. Miners generated "5 quintillion 256-bit cryptographic hashes every second" in June of 2017, and that the estimated energy usage in that case was around 500 megawatts; miners are simultaneously hashing nonces for the reward of cryptocurrency and this is where the power drain comes from [23]. Miners compute hashes until they find a hash digest with a specific number of leading zeroes, but if these hashes are found too quickly, the algorithm on the network increases the number of leading zeroes, and effectively the mining difficulty, required for a valid block which causes miners to try more and more nonces with the hash function to obtain a match [23].

I. SUMMARY

This chapter discussed components and the history of the blockchain technology. Core concepts of blockchain technologies were introduced to provide a context for the rest of the study. A couple of use cases for blockchain technologies were introduced to emphasize the utility of the data structure. The next chapter will define the criteria we will use to select a BP for use in the rest of the study.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SELECTION CRITERIA

This chapter will discuss Blockchain Platform (BP) features that should be considered when making a platform selection. “Platform” in this case means a blockchain application that is machine or operating system agnostic. These criteria are meant to aid in choosing a versatile application platform. The following sections name the criterion that we looked at, and the units for those measurements.

A. TECHNICAL SUPPORT AND TUTORIALS

This criterion measures whether the blockchain platform has a technical support infrastructure, such as community forums, or a dedicated help line from the developers that facilitates solving technical problems. Setting up a new system can be difficult, and the process can present errors and problems that might not be reflected in platform documentation. Users can benefit from help with problems from the developers, or others who have more experience with the platform. The focus of this criterion is on the community and developer’s ability to obtain help to solve issues or answer questions. Tutorials on basic setup operations that quickly move the platform to a usable state are also valuable.

Community and developer support platforms could take the form of a forum like Stack Exchange, Stack Overflow, GitHub, or social media sites. Tutorials could be either provided in documentation from developers or may be available from community members. Some platforms could include paid support options, such as the Red Hat Linux support subscription noted on their support page [24].

Ease of access to and availability of a support structure for the platform will dictate scoring. The scale is low (1), medium (2), and high (3). *Low* means that the community is barely developed: the response rate to questions is low, allowing posts to go unanswered or answered very slowly. A low score could also mean that developers have offer limited support options, do not respond to questions aimed at them either for long periods or ever, and that there are few, if any, tutorials for users. *High* means that the community for the platform is mature and that users can seek help from each other in forums, or that developers are able to take the time to answer more in-depth questions and have provided tutorials to cover common questions. *Medium* means one or two of the characteristics of the *high* rating might be present individually, but not all the characteristics are present.

B. LEVEL OF PLATFORM SUPPORT

Platform support corresponds to the frequency and quality of both functional and security updates and patches. A platform that is still receiving new features and security updates from its developer community is valued over one that has stagnated or might not be supported anymore. An example of this is the service that Ubuntu provides for its Linux distribution. Ubuntu releases new server and desktop versions every six months. These include updates to security and applications [25], thus ensuring that the platform incorporates desirable new features. A blockchain that has any sort of update or release system would be valued over one that does not have such a system, or one that has less frequent updates or releases.

The level of support will be judged based on the frequency of updates that the platform developers report from their websites. If a release history is available, this will be used to score the blockchain platform. The scale is for this criterion is also represented as low (1), medium (2), high (3). Low means that the platform is not updated often, once every few months or less frequently, or it seems that development of the platform has stopped. Medium means that the platform is updated at infrequent intervals, but new features and security updates are still being made. “Infrequent” means an update to the platform is available between every month to every three months with no set update routine. High would mean that the updates to the platform are both reliable and routine or frequent for both security and functional updates. “Frequent” means every month an update is available, and “routine” means the platform developers have specified a reliable update schedule.

C. PROPRIETARY OR FREE AND OPEN PLATFORM

Are the platform source code and binaries freely usable, or, if the platform is proprietary, it must they be purchased. This criterion is disjoint from licensing considerations, which are discussed in Section D of this chapter. A platform for which code and binaries are freely available, and that may be freely modified would be ideal for the testbed. This freedom would allow custom functionality to be added or for existing, but superfluous, functionality to be eliminated.

This is a binary option: the platform is either free and open and does not require license purchases to use core functionality, or it does require a license to be purchased. Appropriately, the scale for this will be free and open (1) and proprietary (0). Even though this is a binary choice, being free and open is seen as positive when setting up the platform for use in this study. If a platform requires a subscription to obtain patches, updates and support, for example, as stated in the Red Hat subscription model page [26], then the platform will be rated (0) for proprietary. For our testbed, platforms that require no form of payment are preferred.

D. PLATFORM LICENSING

This criterion focuses on the kind of license associated with the platform. This is important because different software licenses place different requirements on the users of that software. Some licenses allow free use of the software and source code and allow developers to publish modified code [27]. Other licenses, like for proprietary software, require management to make sure that users are making use of the software and that the organization does not overpay for unused software licenses [28].

This criteria entry is the name of the type of license for the platform. Although licensing will be considered when selecting a BP, it will not be used to directly select or rank a BP. Instead of quantifying licensing, the name of the license will be provided. For proprietary licenses, if the license type or name is not immediately clear, then the value will be “Proprietary: name of company,” thus recording that this platform has a license for which payment may be required.

E. DATABASE CAPABILITY

This criterion addresses whether a blockchain platform integrates, or can integrate with, a database system. “Integration” in this case means that the BP allows users to view contents of the ledger through a database system. This database system should contain the ability to export data from the ledger. If storage is a limitation for a blockchain application, then an auxiliary database would attractive.

Since a platform either does or does not integrate, or can integrate, a database system, this is a binary choice. The values will be either no (0) for not integrating a database system or yes (1) for having database capability or integrating a database by default. If database integration is a feature the developers will introduce in the future, this is considered equivalent to non-database integration and will receive a no (0) rating. Database capability is seen as a positive because it presents more application possibilities. If no information about database capabilities can be found from appropriate BP documentation, then it will be assumed that database functionality is not supported and the criterion will be set to no (0) for that platform.

F. OPERATIONAL MODES: PRIVATE OR PUBLIC AND PERMISSIONED OR NON-PERMISSIONED

The type of membership, or operation mode, the platform supports is important because it affects how the platform can be used as an application. A blockchain could permit users to configure the system to support various membership policy options. A public blockchain allows anybody to join and become a participant on the network, a private blockchain can only be joined by invitation [29]. A permissioned blockchain only allows approved entities or members to use any blockchain functionality [30]. A permissionless blockchain will allow anybody to join and start using the system without being approved and identities are not necessarily associated with participants [30]. These operational modes are important because they govern how the users on a system are represented and what powers they have.

Two separate criteria: public-private and permissioned-permissionless, present binary choices. For the public-private criterion, we will determine if the BP has the capability of being private. For the permissioned/permissionless criterion we will see if the BP can be run in a permissioned mode. The values for both are either yes (1) or no (0). *Yes*, means that the platform has the capability to be run privately and permissioned for each criterion respectively. *No* means it does not have the capability for a criterion respectively.

G. OPERATING SYSTEM OR SOFTWARE PLATFORM ENVIRONMENT

This criterion is the operating system or virtual environment that the blockchain platform depends upon. This criterion specifies the operating system or software suite that must be installed on a machine for the BP to execute correctly, or at all. This criterion does not address technical specifications of the BP itself. The operating system is important because the platform must be compatible with existing systems if it is to be used for an application across a network. If the blockchain technology can only run on a proprietary operating system such as Windows, then a license must be purchased for each node on the network. This will add to the cost of launching and operating a blockchain system. On the other end of the spectrum, if the BP runs on an open source platform such as a Debian-based Linux distribution on bare metal, in a virtual machine or in a Docker image, purchases of operating system and virtual machine licenses are not required. This can reduce the cost of setting up a blockchain network.

This criterion cannot be quantified numerically so we will list the operating system or platform environment the BP executes on. If a blockchain technology can execute on more than one operating system, then the value will be “multi-platform.” To be “multi-platform,” a BP must be able to operate on Windows, Mac OS, Linux, and Docker. It is possible that every BP surveyed will be multi-platform. If that is the case, it will be mentioned in the platform selection discussion that this criterion was not used in the selection process but was taken into consideration.

H. SUPPORTED APPLICATION LANGUAGES

The application support language of a BP is an important consideration because choice of language could affect the developers who work on the applications. A domain-specific language has an internal form and an external form, which are interpreted by a general-purpose language [31]. Internal form domain specific languages could include CSS and regular expressions, since they are parsed by general purpose languages [31]. An external form, they mention, could be like a form of application programming interface (API) with a specific syntax that is present in a general-purpose language [31]. In contrast, a general-purpose language is used in a wide variety of situations and for different problems [32]. General purpose languages include C/C++, Java, JavaScript, and others like them [32].

We assume that a general-purpose language is more diverse in and versatile in application development, and that a wider range of developers will be skilled with it. Using a domain-specific language can increase the learning time of using a BP [33]. For blockchains, domain-specific languages such as Solidity on the Ethereum platform are designed to keep execution of smart contracts deterministic [33]. Determinism can be defined as being the "order-execution-architecture" of consensus where transactions are sent to every peer, and then executed sequentially by those peers [33].

The BP that we ultimately choose must support the use of one or multiple general-purpose languages. Platforms that use domain-specific languages will be scored lower. This is a binary choice, so the scale will be: domain specific (0) and general-purpose (1). There are advantages and disadvantages for both general-purpose and domain specific languages, but, for this study, we prefer general purpose languages because they are more compatible with the software engineering process than domain specific languages and this is a quality we think would be valuable for government and military use [34].

I. PLATFORM IMPLEMENTATION LANGUAGE

The programming language that a BP is implemented in can be different than its application language. The implementation language is the language that the developers used to build up basic features of the BP. We will avoid esoteric languages because they might not be practical in a production environment and could have been developed as a joke [35].

These are general-purpose languages that we would like our candidate BP to be implemented in. We will use the TIOBE index of computer languages to determine how common a language is [36]. We will interpret a higher rank to mean that a language is more common and in wide use. The scale is common (3), less common (2), esoteric (1). Common languages are C/C++, Python, Java, and Golang. They are in the top twenty on the TIOBE index. We will consider less-common languages to be ones that are below the top twenty from the index, which lists one hundred languages. Esoteric languages will be any language not listed on the index.

J. DOCUMENTATION FOR CODE

Documentation for a platform is important for understanding the capabilities and features available when designing applications. An API is defined as a set of standard functionalities available to programmers and application designers so that they do not have to start from scratch [37]. Application developers should be able to reference functions and API calls so that time is not wasted recreating features that already provided by the platform.

For this criterion, we will measure the availability and ease of navigation through the code's documentation Key features include explanations of function or API behavior including data structures expected as input and output, and that there is enough documentation for application designers. The scale for this will be none (0), minimal (1), moderate (2), and high (3). It is unlikely that developers would fail to publish documentation for a platform that they want others to use, but if documentation cannot be found, the platform will be scored none. Minimal level documentation will be assigned to platforms that have hard-to-find or difficult to interpret documentation, or documentation that is hard to navigate. A moderate level of documentation will correspond to documentation that is easy to find and will provide coverage of important code features. A high score for documentation is reserved for platforms that provide easy to access and navigate documentation, and that covers all the features available to developers for the current version of the platform.

K. DOCUMENTATION FOR ADMINISTRATION OR MAINTENANCE

Documentation administration and maintenance is important because a BP, much like any other system used for a long time, requires software updates, user management, and administrator management. This criterion measures the quality and amount of documentation for those purposes.

Much like the documentation for code, ease of access, availability, and coverage of features factor into the scale for this criterion. The scale is none (0), minimal (1), moderate (2), and high (3). None means that we were unable to find documentation for managing the platform. Minimal means that documentation exists, but it either does not cover the features well or is still in development. A rating of moderate means that coverage of most features is provided, and that the documentation is accessible and easy to navigate. A high rating is given to documentation that covers all the features of the current software, and that is accessible and easy to navigate.

L. PLATFORM MATURITY

It is important to know when a platform was announced so its level of maturity can be considered. A maturity measure can also help to determine if a platform is still active or not. For example, if the platform was announced on a certain date but has not been updated for a long time, then this platform is probably not being supported anymore. If the platform is still being regularly updated, it is a better candidate for selection for the testbed.

The maturity level will be based on the years since the announcement of the BP and, if available the number of years it has been active. The years in an active state will be determined from the update and version history of the platform. If updates have not been made for more than a year, the platform will be deemed inactive and it will only be as mature as the number of years from announcement to final update. The scale is: low maturity (1), medium maturity (2), and high maturity (3). A low maturity rating means that the platform is less than one year old since its announcement date, medium maturity applies to platforms one to three years old, and high maturity is assigned to platforms that are greater than three years old.

M. COMMUNITY ACTIVITY LEVEL

This criterion measures the size of the community surrounding the platform. A community involves communication between users and developers about platform news, and how they use the platform. The users could have some say in the new features to be developed, thus creating an aspect of user participation. Additionally, the users and developers might use the community to organize events based around the platform.

A platform's level of activity in the community could be an indicator of the platform's lifespan and status. More people using and talking about the system with each other and to the developers means that there is incentive to keep the platform active, and that its broad user base is an indicator of its stability: both positive factors for selecting a BP for our testbed.

This field will report the level of activity of the community associated with the platform. A community could include social media platforms; exchange or forum sites, like stack overflow; or the platform website's own forum. The rating scale is active (1) and inactive (0). Active means that the community activity indicates that the platform is still in use and that developers and users are in communication with each other. Inactive means that developers and users may have once communicated with each other, but no longer do so, or that a community never formed around the platform. "Inactive" for this criterion is defined as a platform that has not had any community updates or activity for six months or more.

N. SAMPLE APPLICATIONS

Sample applications available for the BP can be used as learning examples and templates for developing new applications. These would take the form of either code that can be modified or executables to demonstrate various features. Sample applications that come from users other than the platform developers could also be helpful, to see what can be built with the platform.

This criterion will be measured by the volume and availability of sample applications for a given BP. Sample applications should be easy to find, and there should be a diverse set of applications in the context of the BP. The rating scale will be low (1), medium (2), and high (3). A low rating means that a small number of sample applications is available: little to none are provided by the developers or users of the platform. A medium rating means that the developers provide a few key sample applications and other users regularly create sample applications. A high rating means that diverse set of sample applications exists that helps drive development, and that users create and share applications on a centralized public forum, or on GitHub.

O. CONSENSUS ALGORITHM

The consensus mechanism of a BP can determine the amount of computation and power required of nodes on the network. Because of the drawbacks of PoW consensus described in Chapter II, this consensus mechanism will be avoided for use in this study. We avoid heavy computation because we require the platform to run on machines that were not built specifically for cryptocurrency mining. We will focus on consensus mechanisms that are less computationally intense.

For each platform surveyed, we will determine if the consensus method is heavily reliant on computation. At the same time, we want a platform that will validate transactions on the network in a satisfactory manner. This scale will measure the emphasis a BP puts on validation mechanisms not purely determined by brute-force computation over the emphasis on brute-force computation. The scale will be high (3), medium (2), and low (1). High means that algorithms that provide validation prioritize non-brute-force computation mechanisms over brute force computation mechanisms, but enough computation is present to validate transactions on the network. This level is the preferred because we assume less intense computation will result in less power consumption, which was mentioned in Chapter II. Medium means that non-brute force and brute-force computation mechanisms have an equal emphasis. Low means that the emphasis on non-brute force validation mechanisms is lower and the emphasis on brute force computation mechanisms is higher and approaches the level of PoW computation.

P. MINIMUM IMPLEMENTATION NETWORK SIZE

The minimum network size of a platform is important because it affects the setup cost of using the BP. If the number of nodes required for a network to be operational is very high, the platform cannot be used with small network configurations, which could be limiting for applications. That said, we assume that every blockchain network reviewed requires at least two nodes.

It is possible that information regarding the minimum network size might not be available until a platform is set up for use. If any information regarding this criterion is available, then it will be considered and documented. Otherwise, this information will not be included in the selection matrix in.

Q. SUMMARY

This chapter defined the criteria that we will use for selecting a BP. When possible, a metric for each criterion was introduced, and additional considerations were discussed. The next chapter will discuss the process of selecting the BP and will discuss the platform chosen as well as several strong contenders.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. BLOCKCHAIN PLATFORM SELECTION

The blockchain platforms (BPs) reviewed in this chapter were found on the curated list “awesome-blockchain” on GitHub [38]. Blockchain platforms were chosen for review if their descriptions indicated a relatively versatile platform rather than one that mainly focused on tokens or cryptocurrency. Next information relevant to the criteria from Chapter III was gathered about the platforms from the relevant websites. Documentation websites were sought out, and usually included a customized documentation site or a wiki page for a GitHub repository for the blockchain platform. Advertised features from the main websites were useful clues for digging deeper into the functionality of the blockchain platforms.

The results of this process were the ranking of twelve blockchain platforms based on the metrics discussed in Chapter III. Numerical scores were given to the blockchain platforms based on the information gathered about each of them. The metrics, and interpretations of those metrics are particular to this study. Capabilities and features that were thought to be useful for the BP’s potential use were emphasized.

This chapter discusses the selected and other high scoring BPs. The discussion of each blockchain platform includes a summary of the features that either contributed to its selection or its dismissal. In this chapter the names of **criteria** are written in boldface. Both quantitative and qualitative criteria were individually given scoring metrics in the form of numerical values. The rank of each blockchain platform is computed as the sum of these values. The numerical criteria are: **Tech Support**, **Platform Support**, **Open Platform**, **Database Capability**, **Permissioned**, **Private**, **Application Support Language**, **Implementation Language**, **Documentation for Code**, **Documentation for Administration and Maintenance**, **Platform Maturity**, **Community Users and Developers**, **Sample Applications**, and **Consensus Mechanisms**. The non-numerical criteria are: **Licensing**, and **When Introduced.** One criterion, **Platform (OS and Software Environment)**, was a uniform “multi-platform” across all the platforms reviewed, so it was not regarded in the selection process and removed as a criterion. Table 1 and Table 2 show the selection matrix for the blockchain platforms considered.

Table 3 and Table 4 contain additional information and factors that have no numerical values but were taken into consideration during selection.

Table 1. Selection matrix results

	Hyperledger Fabric	NXT	Waves	Ethereum	Multichain	Stratis
Ranking	29	27	26	25	25	24
Tech Support	high (3)	high (3)	high (3)	high (3)	high (3)	high (3)
Platform Support	medium (2)	high (3)	high (3)	medium (2)	high (3)	high (3)
Open Platform	open (1)	open (1)	open (1)	open (1)	open (1)	open (1)
Database capability	yes (1)	yes (1)	yes (1)	no (0)	no (0)	no (0)
Permissioned	yes (1)	yes (1)	yes (1)	yes (1)	yes (1)	yes (1)
Private	yes (1)	yes (1)	yes (1)	yes (1)	yes (1)	yes (1)
Application Support Language	general purpose (1)	general purpose (1)	general purpose (1)	domain specific (0)	general purpose (1)	general purpose (1)
Implementation language	common (3)	common (3)	less common (2)	common (3)	common (3)	common (3)
Documentation for Code	high (3)	high (3)	moderate (2)	high (3)	moderate (2)	high (3)
Documentation for Administration/ Maintenance	high (3)	high (3)	moderate (2)	high (3)	high (3)	none (0)
Platform Maturity	high (3)	high (3)	medium (2)	high (3)	medium (2)	medium (2)
Community: Users and Developers	active (1)	active (1)	active (1)	active (1)	active (1)	active (1)
Sample Applications	high (3)	medium (2)	low (1)	medium (2)	medium (2)	medium (2)
Consensus Mechanism	high (3)	low (1)	high (3)	medium (2)	medium (2)	medium (2)

Table 2. Continuation of selection matrix

	Decent	Counterparty	Factom	Monax	Expanse	HydraChain
Ranking	24	24	23	22	21	20
Tech Support	high (3)	high (3)	high (3)	high (3)	low (1)	low (1)
Platform Support	high (3)	medium (2)	high (3)	high (3)	high (3)	medium (2)
Open Platform	open (1)	open (1)	open (1)	proprietary (0)	open (1)	open (1)
Database capability	no (0)	no (0)	no (0)	no (0)	no (0)	no (0)
Permissioned	no (0)	no (0)	no (0)	yes (1)	yes (1)	yes (1)
Private	yes (1)	yes (1)	yes (1)	yes (1)	yes (1)	yes (1)
Application Support Language	general purpose (1)	domain specific (0)	general purpose (1)	domain specific (0)	domain specific (0)	general purpose (1)
Implementation language	common (3)	common (3)	common (3)	common (3)	common (3)	common (3)
Documentation for Code	high (3)	high (3)	moderate (2)	moderate (2)	minimal (1)	moderate (2)
Documentation for Administration/ Maintenance	high (3)	high (3)	high (3)	moderate (2)	high (3)	moderate (2)
Platform Maturity	medium (2)	high (3)	low (1)	medium (2)	high (3)	medium (2)
Community: Users and Developers	active (1)	active (1)	active (1)	active (1)	active (1)	active (1)
Sample Applications	medium (2)	medium (2)	medium (2)	medium (2)	medium (2)	low (1)
Consensus Mechanism	medium (2)	medium (2)	medium (2)	medium (2)	low (1)	medium (2)

Table 3. Additional platform information and factors

	Hyperledger Fabric	NXT	Waves	Ethereum	Multichain	Stratis
Licensing	Apache License 2.0	Jelurida Public License version 1.1	Apache License v2.0	GPL v3.0 and MIT License	GPL v3.0	MIT License
When Introduced	12/ 1/2015	11/24/ 2013	6/7/2016	7/30/ 2015	9/10/2016	6/14/2016

Table 4. Continuation of additional information and factors

	Decent	Counterparty	Factom	Monax	Expanse	HydraChain
Licensing	GPL v3	MIT License	MIT License	GPL v3	GPL v3	MIT License
When Introduced	1/1/2016	1/2/2014	11/17/ 2017	9/8/2015	6/1/2016	1/15/ 2016

A. THE SELECTED BLOCKCHAIN PLATFORM: HYPERLEDGER FABRIC

Hyperledger Fabric (HLF), from initial investigation, appears to be a versatile decentralized application platform. Development on this platform does not appear to be limited to cryptocurrency or token related blockchain operations. The architecture and capabilities of HLF will be discussed in Chapter V.

Based on the considered blockchain selection criteria discussed in Chapter III, HLF was deemed the most appropriate choice. HLF focuses mainly on the distribution of the ledger, smart contracts, and membership. Several features of HLF differentiate it from other blockchain implementations.

The first, public key infrastructure support, although not discussed in Chapter III but discovered upon further investigation, was found to be a valuable capability. The documentation provides instructions and details for setting up certificate authority servers for use within the blockchain network. Other BPs have mentioned use of private keys, but none of them mentioned or detailed setting up a custom CA server or using existing CA certificates for the blockchain network. This could be an attractive capability for U.S. military because it would mean that current CA systems could be integrated into the HLF platform.

The second feature is HLF’s application programming language. Smart contracts for Hyperledger Fabric are implemented with the Go language and are called *chaincode* [39]. The Go language, and how it fits into HLF, will be discussed in Chapter V. The developers plan to support general purpose languages such as Java in the future [39]. This sets this platform apart, in that it does not require its own domain-specific language for transactions. Requiring developers to learn the Go language for development is acceptable for this testbed. The advantage here is that the language and compiler will be better supported. When the use of multiple languages for smart contracts is implemented, it could cut down on training costs and learning periods for developers.

B. PLATFORM RANKING

Table 5 is a chart rating the blockchain platforms according to the score they were given during the selection process. A high score means that the platform was very appropriate for the testbed and is a platform that was chosen or that could be a runner up for installation on the testbed. Platforms rated at a score of 25 and above could be suitable for the testbed. Those found to be focused mainly with cryptocurrency, instead of a broader set of applications are ranked less favorably. Platforms with ratings of 24 or below were considered unsuitable for the testbed. For example, a platform may have been found to focus too much on tokens or cryptocurrency and mining.

Table 5. Platforms ranked by score

Platform	Rank
Hyperledger Fabric	29
NXT	27
Waves	26
Ethereum	25
Multichain	25
Stratis	24
Decent	24
Counterparty	24
Factom	23
Expanse	22
Monax (formerly Eris)	22
HydraChain	20

C. REJECTED PLATFORMS

This section discusses the platforms that were rejected. Analysis of features deemed unsuitable for use in the testbed is provided.

1. Ethereum, Counterparty, Monax, and Expanse

These four platforms are mentioned together because they extend Ethereum or interact with it in some way.

The Ethereum platform mainly deals with smart contracts. These contracts can be used to trade the cryptocurrency that the platform uses or other user-specified transactions. The Ethereum platform is attractive for its smart contract-oriented transactions, but, because it only supports programming smart contracts in Solidity, it was not selected for the testbed.

Counterparty interacts with the Bitcoin and Ethereum Blockchains [40]. Monax, Expanse and Counterparty rated lower than Hyperledger Fabric and Ethereum overall, therefore we decided that they were not the best fit for the testbed.

Monax first started as a fork of Ethereum but then pivoted into its own platform allowing the previous work to be converted into the Hyperledger Burrow project [41]. At the time of our report, the Monax platform had not yet pivoted, and the column in the selection matrix does not reflect its current state. Currently, Monax focuses on using permissioned blockchain for operations in the legal space as a contract management system [42]. As currently described, Monax seems to be a solution system for legal products or asset management. While this could be useful, it appears that the ability to develop custom applications with the platform is limited compared to the chosen platform of HLF

Expanse is the first stable fork of Ethereum, and according to their introductory documentation, the platform was designed to be democratically controlled and more efficient than Ethereum [43], [44].

2. Waves

The interface to the Waves platform is a web API [45]. This platform can be set up on private nodes and accessed through the API and its command line interface; however, both its documentation and website indicate that cryptocurrency is its focus [46]. Although other applications could be developed, it would be necessary to ensure that the mechanism does not engage in mining.

3. NXT

NXT is another cryptocurrency platform that offers the ability for users to develop other applications. This would not be a problem, but like the Waves platform, it is not clear how much of the development will revolve around cryptocurrency and mining. This platform could possibly be used for the testbed, but, to use it in a non-cryptocurrency-based configuration, could require more configuration effort than for other platforms. The website for NXT does not mention smart contracts, only asset management and monetary systems [47], [48]. Thus it is likely to have less utility in a government network environment.

4. Multichain

The Multichain BP has permissions, and the ability to create a private network of nodes. These are valued features for a blockchain for a private network, but, according to posts on the company's website, this platform is does not seem to be focused on or support smart contracts [49]. It is likely that considerable effort would be required to set the platform up for ledger and transaction logic without the use of a cryptocurrency.

5. Stratis

This platform has been developed in C# for the .NET framework. Stratis allows developers to use the platform for either cryptocurrency, or smart contract-related applications. The platform provides an interface to Bitcoin functionality in the form of NBitcoin, a popular C# library for the .NET framework [50]. At the time of our evaluation, smart contract functionality for Stratis was an alpha release [51]. Because Stratis platform was not ready to support smart contracts, it is ranked below HLF and Ethereum.

6. Decent

The Decent blockchain advertises decentralized applications. The Decent documentation shows that mining is a part of the consensus algorithm for the platform, and anonymity is its main guarantee [52]. Anonymity and mining are not good attributes for a blockchain application used by the DoD. The characteristics of the platform do not make it suitable for selection.

7. Factom

The Factom platform interacts with the Bitcoin blockchain to “take advantage of the security of Bitcoin’s hashrate” [53]. Since the platform leverages the Bitcoin blockchain, it is not suitable for this study.

8. HydraChain

This platform seems to be the least mature of those surveyed and is not appropriate for use on our testbed, or eventually in an operational network. The website associated with the platform has a brief description of HydraChain and points to a GitHub repository and its wiki pages [54], [55]. Technical support appears to require contact with a member of the HydraChain team, which may be problematic. This platform too immature for consideration.

D. SUMMARY

This chapter discussed the selection process for the BP that was used for the testbed. The platforms were ranked and HLF was chosen. The scores for each criterion were presented in a selection matrix. Factors favoring HLF was discussed. Factors that resulted in lower rankings for the remaining BPs were discussed. The next chapter will describe the architecture of HLF, the selected BP.

V. THE HYPERLEDGER FABRIC SOFTWARE ENVIRONMENT AND ARCHITECTURE

This chapter is intended to explain the Hyperledger Fabric (HLF) environment used in the testbed. It includes details of the software suite in use by HLF for application development. The consensus method for HLF will also be discussed. The environment that this study uses is Ubuntu 18.04, a Linux distribution, on a virtual machine using the VirtualBox software. Detailed instructions for setting up the software and development environment on a virtual machine are provided in Appendix A. In our discussion, *utility names* in HLF will be written in italics. **`Entity names`** will be written in the boldfaced typewriter font. ***Section names*** will be bold and italicized. **File names** will be written in bold.

A. COMPONENTS OF THE HYPERLEDGER NETWORK

This section will detail the software necessary for an HLF blockchain network. The software required for operation includes: Docker, Docker Compose, CouchDB, the compiler associated with the Go language, and Node.js.

1. Docker

The Docker software suite provides an environment that supports lightweight virtual machines, called *containers* [56]. It is an open platform for developing and running applications in separated environments [57]. According to the OpenSource website [56], Docker containers are used by developers to bundle an application, its libraries and dependencies. Instead of executing applications in separate full virtual machines, multiple Linux-based applications can be supported in individual containers on a single Linux machine. The Docker design is useful for setting up nodes quickly. They can be used for application development and testing, especially in the early stages of development.

In combination with Docker Compose, the Docker software is used to sets up the containers that will operate the blockchain network. These containers act like separate computers complete with their own set of associated HLF utilities.

2. Docker Compose

Docker Compose supports the setup of multiple containers in an application environment. Using configuration files in the YAML format, a user can configure containers with environment variables, ports for services, and container names [58]. The YAML format will be discussed in the ***YAML Configuration Files*** section of this chapter. Docker Compose can be used to put together an application where the specifications for each container and service will dictate the overall behavior of the application. The configuration file format used by Docker Compose is also used by the utilities, or platform functionality commands, provided by the Hyperledger Project. Those utilities and the configuration file format will be explained in the ***Hyperledger Fabric Utilities*** section.

3. CouchDB

CouchDB is an Apache-based database software that can store binary and JSON (JavaScript Object Notation) data [59]. Couch DB is one option for maintaining state in the HLF network. The HLF documentation on Couch DB states that, by default, a database called LevelDB is used to store "key-value state" across the network [60]. This key-value state is any data that can be modeled in the chaincode, making these databases effectively storage for the ledger state. CouchDB offers the ability to store any data generated from chaincode that can be represented as JSON data, and to then query for that data. JSON according to RFC 8259, is a human-readable, language-independent file format for data organized in attribute-value tuples [61].

4. Golang

The Hyperledger code base has been written in Golang. Development of this language began in 2007 to address software infrastructure issues at Google [62]. The Go language shares similarities with the C and C++ programming languages [62]. This high-level language is similar enough to Python and C/C++ that the learning time is short for those conversant in either of those languages.

5. Node.js

Node.js is a framework extension of the JavaScript language. The platform was developed as an asynchronous runtime environment for network applications; it takes advantage of event-driven programming, concurrency, and was influenced by the Ruby and Python languages [63]. A network application in this case is defined as an application that communicates using the network as the underlying infrastructure. How this platform is used in HLF will be explained in the chaincode section.

6. Chaincode

Smart contracts in HLF are handled with chaincode. Chaincode, HLF's name for smart contracts, can be developed in Golang or Node.js, but the HLF documentation states that the developers, at the time of writing, will add the capability to write chaincode in additional general-purpose languages such as Java [39]. This design will allow developers to choose the programming language for chaincode application development most appropriate to their company or organization.

Currently, people who use the HLF platform, must choose between Golang or Node.js as the language for writing chaincode. The two languages are different in design as well as perceived difficulty. Ease of use is a factor in language selection. The HLF documentation is neutral regarding these languages, but it would make sense for a programmer to gravitate towards whichever language seems easier to work in. Snellinckx argues that Node.js could be easier to use by more developers than Golang [64].

Another factor in language selection is performance. The Go language compiles to machine code before execution [65]. Conversely, Node.js is an extension of JavaScript, and is interpreted by the Google Chrome V8 JavaScript engine [66]. Compiled interpreted languages represent implementation choices for programming languages. A compiled language is translated to machine code specific to the device it executes on [67]. Any machine that runs an interpreted language needs the interpreter program for that language installed. The interpreter translates the human readable code to an intermediate level that is then translated to machine code [67]. The benefits of this, are that the code is easier to write and modify, and it can run on any machine, albeit slower than a compiled language.

A possible drawback is the absence of a language interpreter for the target instruction set architecture.

Two styles of chaincode development exist for the HLF platform. Called “personas,” Fabric [39] offers two perspectives: “Chaincode for Developers” as the application-oriented development perspective, and “Chaincode for Operators” as network manager and chaincode maintenance perspective. These two personas would apply to different sections within a department that deals with the HLF platform. The definition of the “chaincode for developers” persona is an application developer creating chaincode that runs on the **`peers`** in the network [68]. Tasks for application developers include defining paths to the source code, writing chaincode with either Golang or Node.js, and implementing functions of chaincode using available library code. The operator’s perspective in the HLF “Chaincode for Operators” [69] focuses on managing the chaincode for the nodes on the network. Tasks include installing chaincode onto **`peers`** in the network, upgrading the chaincode on **`peers`**, and instantiating chaincode with keys and values for channels.

B. ROLES OF ENTITIES IN THE NETWORK

This section describes the entity roles and groups on the network such as: **`clients`**, **`peers`** and **`anchor peers`**, the concept of membership and **`organizations`**, **`channels`**, and the ordering service and **`orderer`** nodes. The term node refers to **`peers`**, **`orderers`**, **`anchor peers`**, or any Docker container that has a function in the network.

Figure 3 is a notional diagram of the basic organization of the software stack on the test bed developed for this study.

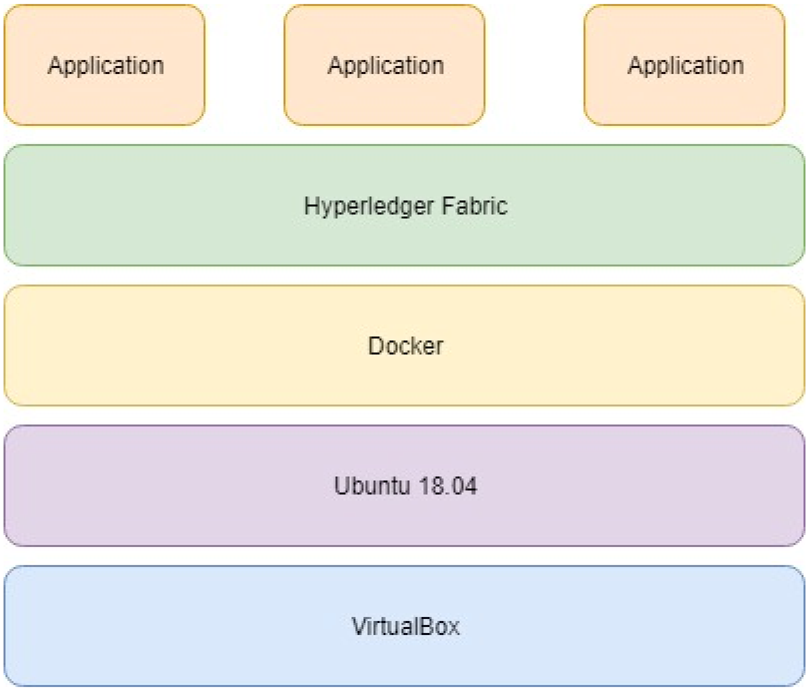


Figure 3. The test bed software stack

Figure 3 starts at the bottom with Virtual Box software that virtualizes Ubuntu version 18.0.4 for the testbed setup. Installed on the Ubuntu virtual machine is the Docker software and HLF software is layered on top of that. Applications reside above the HLF layer. These applications take the form of chaincode and interactions between **`peers`**. It is possible for multiple applications to be supported in one HLF network. Entities in the HLF platform exist within Docker containers, and will be discussed in Figure 4.

1. Clients

`Clients` are associated with the people who use the HLF network application. A **`client`** acts on behalf of a user and must connect to a **`peer`** to conduct read and write operations on the ledger [70]. A **`client`** is an application that interacts with the HLF API to send commands to **`peers`** and run chaincode in the HLF architecture [71]. In this model, the **`client`** nodes are closest logically to the people who use the network, whereas the **`peers`** are a bit closer logically to the architecture's core. In the stack pictured in Figure 3, a **`client`** application would exist at the Ubuntu layer. The **`client`** application could exist within a web browser, or any software that connects with the HLF API to communicate with **`peer`**s.

2. Peers and Anchor Peers

A **`peer`** is any entity that has a ledger and can read and write to it, something that a **`client`** cannot do [72]. **`Anchor peers`** are a special type of **`peer`** on an HLF network. An **`anchor peer`** is a node that facilitates communication between **`organizations`** because a node from one **`organization`** must know the address of one or more nodes in another **`organization`** for communication to occur [72]. **`Peers`** must use their anchor **`peer`** as a middle man to talk to **`peers`** on other organizations. Figure 4 shows the HLF network on top of the Docker layer, this means that peers in HLF are represented as Docker containers. Any number of **`peers`** can exist on the network and execute as Docker containers.

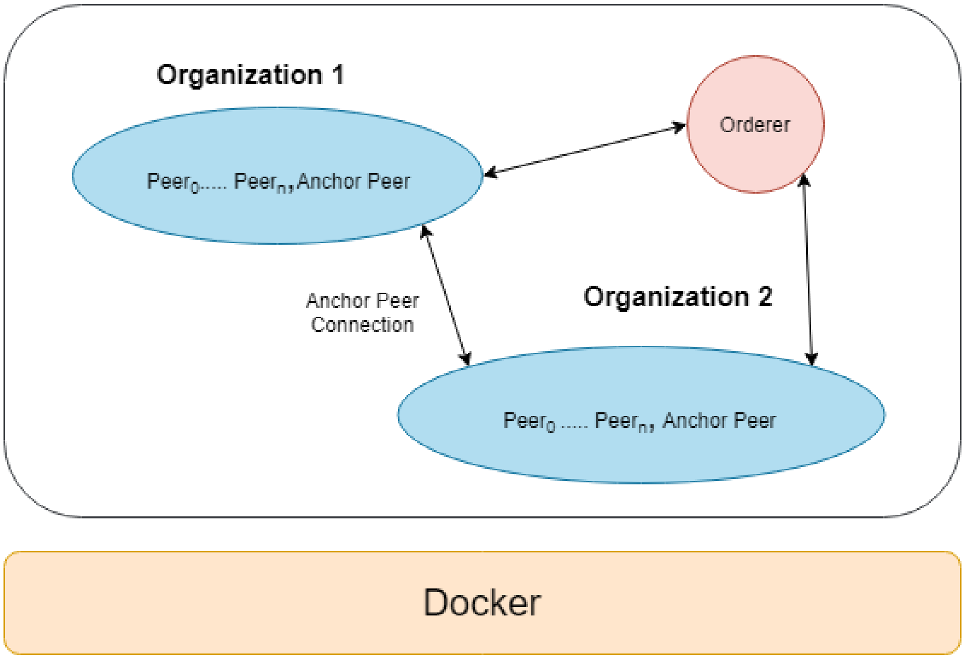


Figure 4. Entities in HLF represented as Docker containers

Figure 4 is a notional diagram depicting how nodes in the HLF network are represented by Docker containers. This is a zoomed in view of the HLF layer just above the Docker layer from Figure 3. Figure 4 is an example configuration of an HLF network consisting of two organizations each with some number of **peer** nodes. Each organization has one **anchor peer**, but it is not a separate entity. One of the **peers** in the group consisting of **peer**$_0$ to **peer**$_n$ is designated to be the **anchor peer** for that organization to facilitate communication between organizations.

3. Membership and Organizations

An **organization** is a collection of **peer**s on the HLF network for which there is a concept of membership. The HLF platform includes an entity called the **Membership Service Provider** (**MSP**). The **MSP** keeps track of certificate authorities (CAs) that define the nodes in an organization [73]. An **organization** can consist of any number of nodes: "as big as a multinational corporation or as small as a flower shop" [73]. What is important is that the members of that **organization** are managed and defined by the **MSP** in the HLF network. The **peer** Docker containers can be logically grouped together into a cohort of entities on the HLF network. Figure 4 shows this grouping of peers into organizations as mentioned in the ***Peers and Anchor Peers*** section of this chapter.

4. Channels

A **channel** in the HLF architecture is a private line of communication within the HLF application network that is composed of **peer**s that have joined the channel, **anchor peer**s, chaincode these **peers** use, the ledger and **orderer** nodes according to the **channel** documentation page [74]. A **channel** can be thought of as an individual ledger, or a separate blockchain with a finite but changeable set of participants, on the HLF network. There can be multiple **channels** comprised of any set of members of the particular network. Although not pictured in Figure 4, channels are represented on every peer. When new peers are become members of **channel**, a copy of the current ledger is shared with them and blocks that result from transactions are distributed to them from then on [74]. The ledgers that represent the **`channel`** are installed on **`peers`**, but the **`channel`** itself can be thought of as a grouping of **`peers`** similar to an **`organization`**.

5. Ordering Nodes and Ordering Service

`Ordering nodes` manage an important mechanism related to the consensus in the HLF network. The ordering service is made up of **`ordering`** nodes that order the transactions on each ledger for each **`channel`** on the network in a "first-come-first-serve basis" [72]. When a **`channel`** is created, the user invokes one *peer* utility command options which notifies **`ordering`** system [75]. This allows the **`ordering`** service to handle multiple channels in one application. The ordering service is treated as a swappable plugin, a selection of defaults come with HLF and they can be changed to other ordering implementations [72]. Figure 4 shows one ordering node that communicates with the nodes from each organization. The **`anchor peers`** allow the organizations to talk to each other, but every node communicates with the **`orderer`** node.

C. CONSENSUS IN THE HYPERLEDGER FABRIC PLATFORM

Consensus in the HLF platform is tied to the transaction flow. Transaction flow is the series of events that occur when a **`client`** starts a transaction within the HLF network. A transaction is described in more detail in the figures below, starting with Figure 5. The definition of consensus, according to the HLF model page of the documentation, is the "full-circle verification of the correctness of a set of transactions comprising a block" [76]. Every portion of the transaction process contributes to consensus in the Hyperledger model. It does not rely solely on a specific algorithm. When a transaction is initiated by a **`client`** application, a request is built that must adhere to the endorsement policy as stated in the assumptions section of the transaction flow portion of the HLF documentation [77]. Figure 5 shows a **`client`** initiating a transaction that builds a request that adheres to the endorsement policy. When the client initiates a transaction, a proposal is sent to the peers that will participate in the transaction [77]. A proposal according to the HLF glossary is a request for reading or writing or adding new data to the ledger [72].

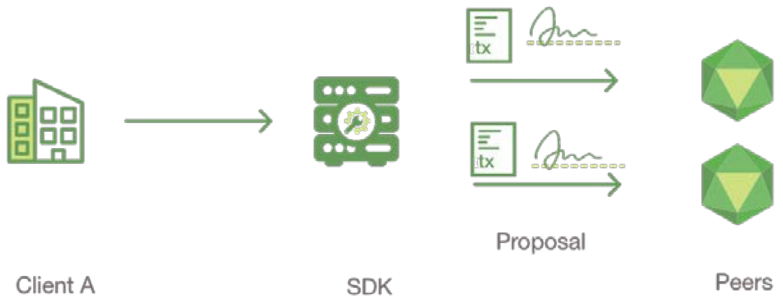


Figure 5. A client initiates a transaction for two peers. Source: [77].

The software development kit (SDK) pictured in Figure 5 is allows the client application to communicate with the HLF platform by taking the generated proposal and wrapping it in the appropriate format for the HLF application network [77]. The SDK in context of HLF is a set of function libraries available to Node.js, and Java that allow an application to interact with an HLF application [78], [79].

As stated in the HLF glossary, an endorsement policy contains a list of the peers that must provide endorsement responses to a transaction request. [72]. A transaction is only valid when the minimum number of endorsing `peers` endorse the transaction request. Returning to the transaction flow process from the transaction flow portion of the documentation [77], the `client` then starts the process of collecting and inspecting signatures from `peers` that endorse the transaction pictured in Figure 6 and Figure 7

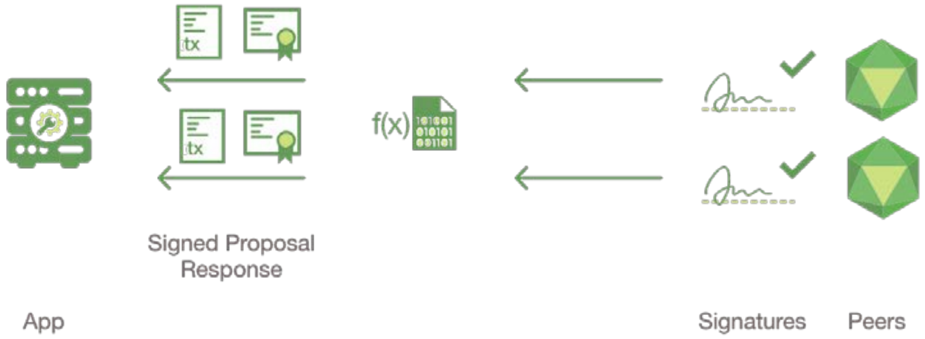


Figure 6. Client application collecting signed proposal. Source: [77].

NOTE: In Figure 6, the icon for the SDK has been given the label “App.” The documentation page that these figures were taken from contains this same inconsistency. The application interacts with the SDK to interact with HLF, so Figure 6 can be interpreted as the application receiving the proposals. Figure 7 can then be interpreted as the application using the SDK to inspect the signatures on the proposals.

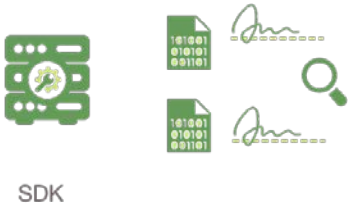


Figure 7. Signatures are inspected. Source: [77].

Figure 6 and Figure 7 show the process of `peers` signing endorsements of a transaction request, which leads to signatures that the `client` application uses to verify and validate transaction proposals. After that, the ordering service orders the transactions into blocks and updates the ledger accordingly as pictured Figure 8 to Figure 10.

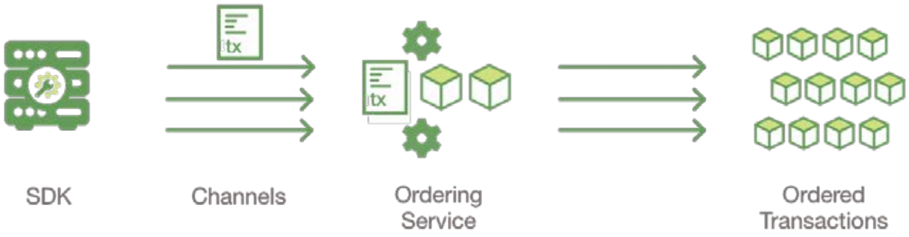


Figure 8. Transaction data sent to ordering service for each channel. Source: [77].

Figure 8 shows transaction data being sent to the ordering system from multiple `channels` on the HLF application network. The ordering system then outputs those transactions in chronological order as blocks.

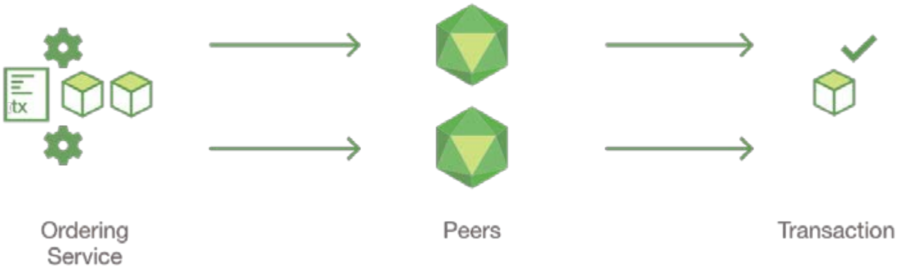


Figure 9. Ordering service delivers transaction blocks to peers. Source: [77].

After the transactions are ordered into blocks, the ordering service delivers those blocks to the **peers** on the HLF application network as pictured in Figure 9.

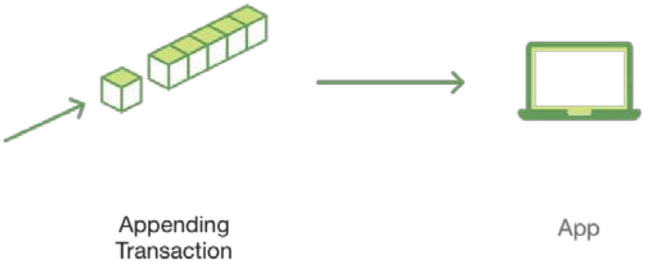


Figure 10. Peers append blocks to their copy of the ledger. Source: [77].

The transaction blocks are then appended to each **peer's** ledger copy as pictured in Figure 10. After this process has completed, the client application will receive both a notification that the ledger has been updated and a notification on the validation status of the transaction [77].

HLF inherently deals with validating membership and identity as well as ordering the transactions on the network. Applications that use HLF do not have to rely on proof of work, proof of stake for consensus, they rely on a cycle of verifications and validations that occur during the transaction.

D. HYPERLEDGER FABRIC UTILITIES

This section will detail the software functionality available in the HLF software environment for operation of HLF applications. This software is comprised of platform-specific utilities that are used for configuration and execution of blockchain operations. The utilities are platform-specific because the installation script downloads the versions of the utility specific to the operating system that the command is executed on. Some of these utilities also use a specific configuration file format to generate output.

The HLF utilities are a suite of tools used to set up an environment for the application network. Setting up the network involves creating communication channels, generating cryptographic material, and defining Docker containers as entities on the network. Some available utilities are: *cryptogen*, *configtxgen*, *peer*, *fabric-ca-client* and *fabric-ca-server*. These utilities are added to a directory which is exported into the PATH environment variable. Thus, users can reference the utilities easily, regardless of their current working directories. The function of each utility within the context of the HLF application framework is explained below.

1. Cryptogen

The *cryptogen* utility provides the ability to stand up a dummy network for testing functionality. This utility generates cryptographic public-private key material for the HLF application framework [80]. This includes certificate authority (CA) certificates, private and public key files for nodes in a network. According to the HLF documentation [80], *cryptogen* is used for testing the environment. This utility uses a configuration file to generate appropriate cryptographic material. A pre-existing PKI, or public key infrastructure, can also be used when setting up the HLF network. In this case, members use their own CA instead of relying on the one generated by *cryptogen* [75]. An existing network that uses HLF would most likely already have a CA for daily network operations.

2. Configtxgen

The *configtxgen* utility is a configuration tool for components on the HLF network. The user can create configuration objects such as `channel` creation transactions, genesis blocks, **`organization`** definitions and **`anchor peer`** transactions [81]. The utility uses YAML configuration files to generate transaction files and can inspect the transactions that have already been generated [81]. These transaction files are not transactions that occur between peers on the network, they are files that specify a transaction in the blockchain network to create certain entities such as **`channels`** and **`anchor peers`** according to the *configtxgen* documentation.

3. Peer

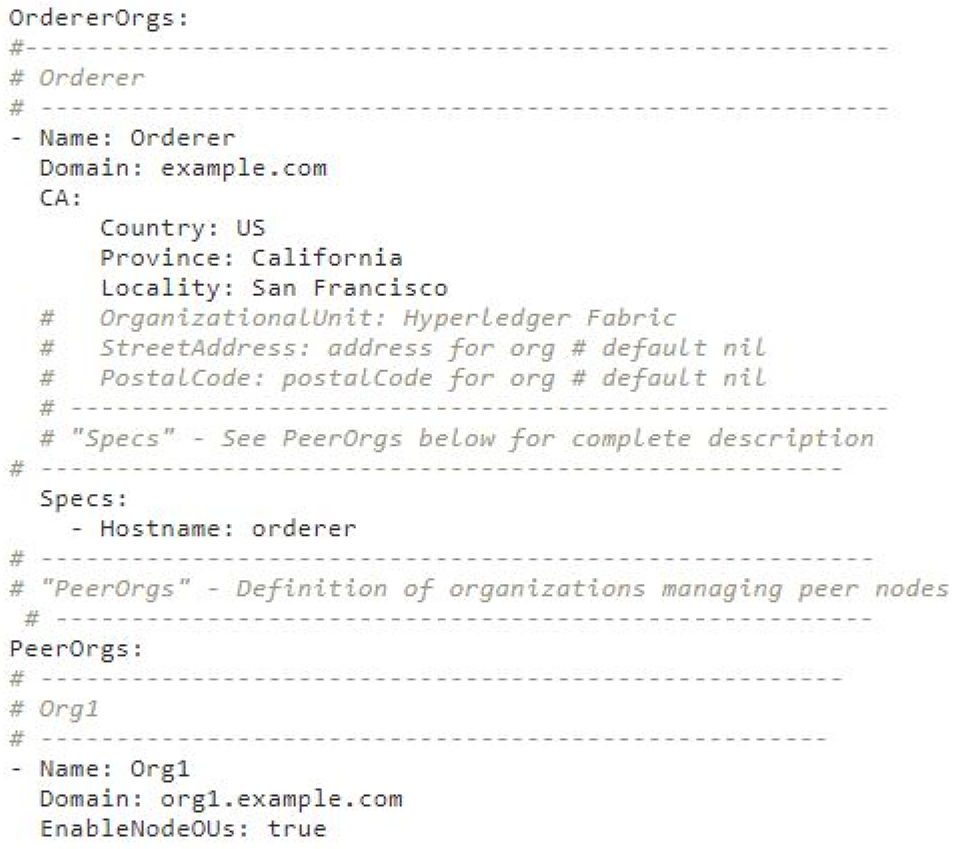
The *peer* utility controls the actions of **`peer`** nodes in the HLF application network. The utility is used to perform **`peer`**-specific tasks [82]. These include adding a **`peer`** to a channel and moving chaincode onto a specific **`peer`**.

4. Fabric-CA-Client and Fabric-CA-Server

HLF supports CA functionality for managing identities on the HLF network. Two commands exist for executing this functionality: *fabric-ca-client* and *fabric-ca-server* [83]. *The fabric-ca-client* command is used to renew and revoke certificates that have been issued, and to manage the identities of entities on the network. The *fabric-ca-server* command is used to start and manage CA servers on the network. The commands have separate syntax pages within the HLF documentation.

5. Use of YAML in Configuration Files

The YAML format is used for configuration information files used by Docker Compose and by multiple HLF utilities. The YAML home page and Ansible documentation websites [84], [85] explain that YAML literally stands for “YAML ain’t Markup Language.” It is intended to be a more human- readable language than XML or JSON. Figure 11 depicts the **crypto-config.yaml** file which describes **`peers`** and organizations so that cryptographic material can be generated.



```
OrdererOrgs:
#---------------------------------------------------------
# Orderer
# ---------------------------------------------------------
- Name: Orderer
  Domain: example.com
  CA:
      Country: US
      Province: California
      Locality: San Francisco
  #   OrganizationalUnit: Hyperledger Fabric
  #   StreetAddress: address for org # default nil
  #   PostalCode: postalCode for org # default nil
  # -------------------------------------------------------
  # "Specs" - See PeerOrgs below for complete description
# -------------------------------------------------------
  Specs:
    - Hostname: orderer
# --------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
 # -------------------------------------------------------
PeerOrgs:
# ------------------------------------------------------
# Org1
# -----------------------------------------------------
- Name: Org1
  Domain: org1.example.com
  EnableNodeOUs: true
```

Figure 11. A YAML configuration file for cryptogen. Source: [75].

E. SUMMARY

This chapter discussed the architecture of the HLF platform. The software dependencies and components of the HLF platform were detailed along with the network entities that create the ecosystem of the network. The utilities used to manage the HLF platform, which are packaged with the platform, were also detailed in this chapter. We gave a high-level overview of how the platform operates behind the scenes when used in a production system. The next chapter will discuss sample applications for HLF provided by its developers. A design for an application that could be used by the military or government will also be proposed in the next chapter.

VI. APPLICATIONS OF HYPERLEDGER FABRIC

This chapter will discuss the sample applications that exist for Hyperledger Fabric (HLF). Also discussed here will be a proposed application for HLF that could be used by the military. The requirements for this application and components involved with an implementation will be discussed as well.

A. SAMPLE APPLICATIONS

Sample applications are scenarios that have been configured and programmed by the HLF developers and packaged into a repository on the GitHub website. This section will describe the applications and their coverage of Hyperledger Fabric functionality. Each application requires similar setup steps to become operational. After that they showcase different functionality and behavior available to users of Hyperledger Fabric (HLF). All applications described here operate on the Ubuntu 18.04 Linux operating system distribution of our test environment.

1. Overview

A common sequence of events is shared by of all the applications. First the network of Docker containers is set up, along with channel artifacts. Channel artifacts are the genesis block file and channel creation transaction files, and anchor peer creation transaction files. Anchor peers and channels, and the creation of the genesis block were described in Chapter V, and the genesis block concept was described in Chapter II. The relevant Docker containers include: containers for peers, the orderer, and management nodes. Second, channels and anchor peers are created on the Docker containers. These represent entities in the HLF application network. Third, chaincode is installed onto the peers, initial values are set and the scenario for the sample application is started.

2. First Network

The ***First-Network*** application provides an introduction to the components necessary for an HLF application network. This application follows the events of the ***Overview*** section of this chapter closely because this application is intended for beginners to the HLF platform. The ***Building your first network*** documentation page associated with this sample application instructs the user to execute the premade scenario before exploring the individual commands within the scenario [75].

The scenario for this application is a basic transfer of value between two keys that are stored in the ledger of one channel. The first task is to generate channel artifacts, which were described in the ***Overview*** **s**ection of this chapter. Since this application is meant to be a demonstration, and is not intended for operational use, the *cryptogen* utility is used to create cryptographic material for the entities on this network. Next, the *configtxgen* utility is used to create the genesis block, and channel and anchor peer transaction files from user-created configuration files. For these sample applications, the configuration files with information relevant to the sample are already provided. These configuration files specify attributes such as the names of channels, the hostnames of peer containers, organization profiles, and the peers are a part of each organization. The *cryptogen* and *configtxgen* utilities were described in Chapter V, under the ***Hyperledger Fabric Utilities*** section.

The second half of the ***First-Network*** sample application involves setting up the Docker containers that represent the network. *Docker Compose* takes a configuration file as input and generates the peer and orderer containers, as well as a command line interface container that is used to manage the other Docker containers in the context of the HLF application [75]. The container that allows for management of the application Docker containers is called the ***CLI*** (Command Line Interface) ***container*** and is generated from the *hyperledger/fabric-tools* Docker image supplied alongside the sample applications by the developers. This container, as described in the documentation for this sample application, contains installed utilities to manage adding peers to channels, and for installing and running chaincode on specific peers. The *peer* utility is used in the operations described next. In the ***Create & Join Channel*** section of the documentation, the ***CLI container*** is used to add peers to channels using their cryptographic material, and predefined channel names as arguments to the *peer* utility. After the peers are joined to their respective channels, the anchor peer transaction files are used to designate anchor peers for each organization. Next, the chaincode is installed on each peer that will be engaging in transactions. After the chaincode is installed application-specific initial values are set. The initial values of this the ***First Network*** application are for transferring data between two entities and for querying the ledger.

All these steps are executed by the scripts available in the sample application, but every step can be executed manually by a user. For example, the network containers can be set up using the Docker and Docker Compose utilities with the provided configuration files. Channel transactions, the genesis block, and anchor peers for each organization can be created manually as well. After the containers are setup, the user uses Docker to enter a terminal session on ***CLI container***. The ***CLI container*** must be attached to using the Docker utility command "`docker exec -it <container name> bash`" to see the terminal screen and interact with the container. The user can then execute commands to add peers to the channel, designate the anchor peers, and install, instantiate and execute chaincode. The scripts that are included in the *scripts* folder, found on the ***CLI container***, show the commands a user could invoke to recreate this sample or execute a custom scenario.

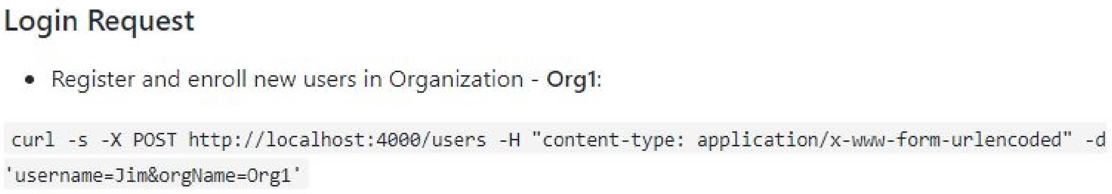
3. Basic Network

Basic Network is a short application that shows basic peer operations in HLF. According to the documentation, the **start.sh** script begins the application, **stop.sh** stops it, and **teardown.sh** removes network artifacts [86]. When **start.sh** is executed, it creates Docker containers for the scenario, creates a channel and joins a peer to that channel [87]. In the beginning of this application, the terminal window shows that one container each is started for a CA server, a CouchDB instance, an orderer node, and a peer node.

4. Balance Transfer

The ***Balance Transfer*** application demonstrates the use of Hypertext Transfer Protocol (HTTP) requests to execute HLF blockchain operations such as registering organizations, creating channels and adding peers to those channels, and installing and instantiating chaincode. HTTP allows a client to request data from the server [88]. In this case, the application uses the *curl* utility to send a request in the form of the URL for the server with parameters appended to the end of it. When the server receives the URL it takes an action based on the parameters and returns the data to the application.

The ***Balance-Transfer*** application starts with the script ***runApp.sh*** which executes a Node.js application to listen for requests on the localhost on port 4000 [89]. The terminal window that this command is executed on then waits for responses from the server. The user then runs the ***testApi.sh*** script next with the parameter ***-l node*** for Node.js chaincode execution or ***-l golang*** for Golang chaincode execution. This script then sends requests to the server for actions such as enrolling users to organizations, creating channels, and installing chaincode. Examples of these command request URL strings are pictured in Figure 12 to Figure 14, from [89]. More examples of commands, and their output, can be found in the README file in the balance-transfer repository.



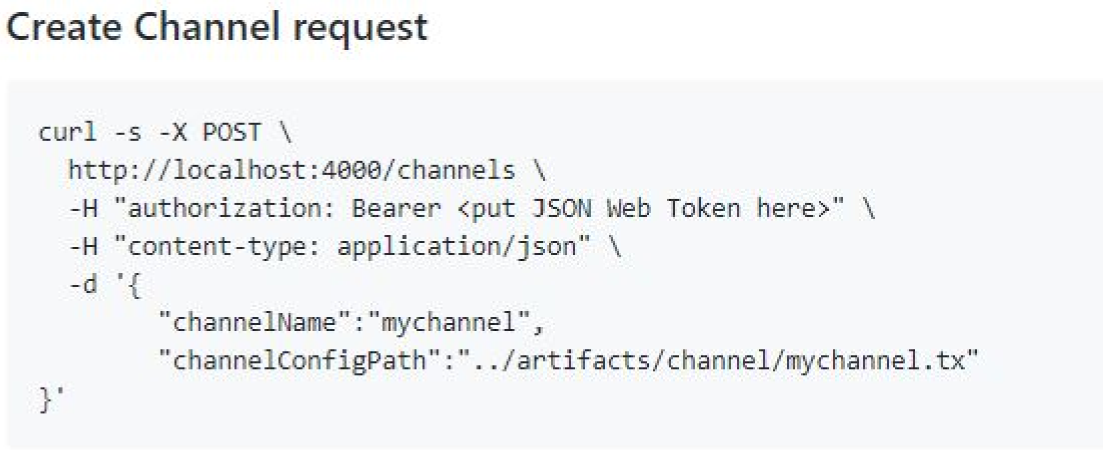
Login Request

- Register and enroll new users in Organization - **Org1**:

```
curl -s -X POST http://localhost:4000/users -H "content-type: application/x-www-form-urlencoded" -d
'username=Jim&orgName=Org1'
```

Figure 12. Command to register new users to organization. Source: [89].

Figure 12 shows the command to add a new user to a specific organization. The parameters include the Organization identifier, or the name of the organization as defined by the YAML configuration files used to start up a Hyperledger Fabric network as referenced in Chapter V.

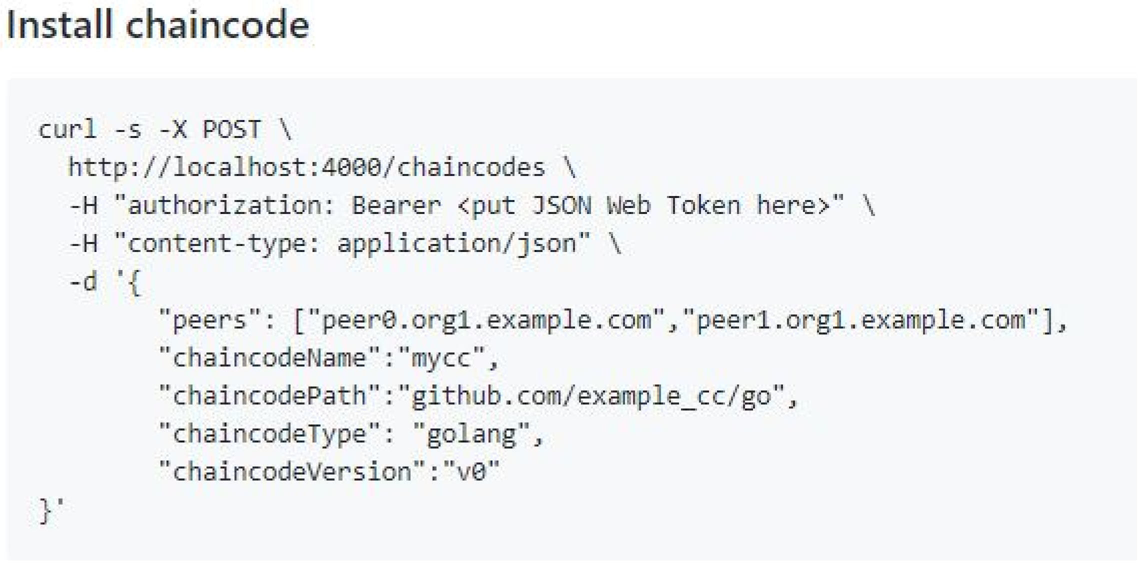


Create Channel request

```
curl -s -X POST \
  http://localhost:4000/channels \
  -H "authorization: Bearer <put JSON Web Token here>" \
  -H "content-type: application/json" \
  -d '{
        "channelName":"mychannel",
        "channelConfigPath":"../artifacts/channel/mychannel.tx"
}'
```

Figure 13. Command to create a channel. Source: [89].

Figure 13 illustrates a request to create a channel on the network. The channel name and path to the configuration file for the creation of that channel are specified in JSON format. JSON was discussed in Chapter V as key value pair data representation.



Install chaincode

```
curl -s -X POST \
  http://localhost:4000/chaincodes \
  -H "authorization: Bearer <put JSON Web Token here>" \
  -H "content-type: application/json" \
  -d '{
        "peers": ["peer0.org1.example.com","peer1.org1.example.com"],
        "chaincodeName":"mycc",
        "chaincodePath":"github.com/example_cc/go",
        "chaincodeType": "golang",
        "chaincodeVersion":"v0"
}'
```

Figure 14. Command to install chaincode. Source: [89].

Figure 14 shows the installation of chaincode onto two peers of Org1. The ***peers*** parameter in the URL show that they are targeting *peer0* and *peer1* to install a chaincode named ***mycc***. The other parameters show the name of the chaincode, the version of the chaincode, and the directory that the source code can be found in.

5. High Throughput

The ***High-Throughput*** application addresses scenarios where transactions happen with a high frequency on the Hyperledger Fabric network. Its documentation states that when an update comes in for a specific key in the network, trying to update the value at that time could cause slowdown of the network, and could create a race condition for the value of that key [90]. A race condition, according to TechTarget, occurs when a shared resource like a file is subjected to a write operation from two different systems or devices at the same time [91]. The result of those operations is then ambiguous and may end up with incorrect output. This sample application proposes a technique of application and chaincode development as a solution to this issue.

This design choice, used to manage race conditions, appears to work best with numerical data. Frequent write operations of numerical data are stored as a row of deltas for a specific key [90]. One use case describes an enterprise bank account that experiences high frequency deposits and withdrawals [90]. The technique describes storing the values of deposits and withdrawals as a series of positive and negative numbers respectively along with the initial value of the account. At some point, when transactions are suspended, a pruning operation is conducted to aggregate the deltas and update the value of the account. The application would process transactions and add deltas with chaincode, and after a certain period, invoke the pruning functions and update the ledger. There are two versions of a prune function: one that is safe, and one that is fast [90]. Fast pruning means that a delta is aggregated then deleted before moving on to the next delta, which is fast, but could lose data in case of error. Safe pruning means that all the deltas are aggregated, backed up, and then deleted, therefore trading speed for caution.

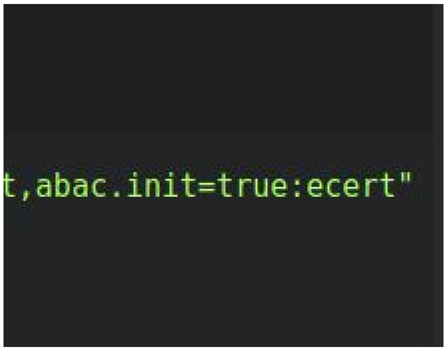
This technique could be extended to accommodate other forms of data.

To run the ***High-Throughput*** application, the documentation instructs the user how to change and leverage the **byfn.sh** script and the **Docker Compose-cli.yaml** file in the ***First-Network*** application for use with the ***High-Throughput*** application [90]. This includes altering the **byfn.sh** script so that it stops short of executing the scenario for ***First-Network*** and editing the configuration file so that the Docker containers use the scripts associated with ***High-Throughput***. Once the application is set up, and the containers are executing, the user enters the ***CLI container*** to execute commands. In our tests, one of the ***CLI container*** scripts did not execute properly. Although the scripts **channel-setup.sh** and **install-chaincode.sh** execute without error, the last script, **instantiate-chaincode.sh,** results in an error with a message that the package for the chaincode ***bigdatacc*** cannot be found. A search through the README file, the chaincode folder for the sample application, the **instantiate-chaincode.sh** file itself, and **channel-setup.sh** and **install-chaincode.sh**, no references to ***bigdatacc*** could be found. This error could be resolved in later versions of the HLF platform. The error was reported to the HLF developer community.

6. Fabric-Ca

The ***Fabric-ca*** example demonstrates the certificate authority (CA) client and server system available in Hyperledger Fabric. The utilities *fabric-ca-client* and *fabric-ca-server* (or *fabric-ca* utilities) are used in this sample. The scenario for this example includes chaincode invocations and transactions between peers on the application network. However, the important feature that this sample application covers is the use of the *fabric-ca* utilities to register and enroll identities on the application network.

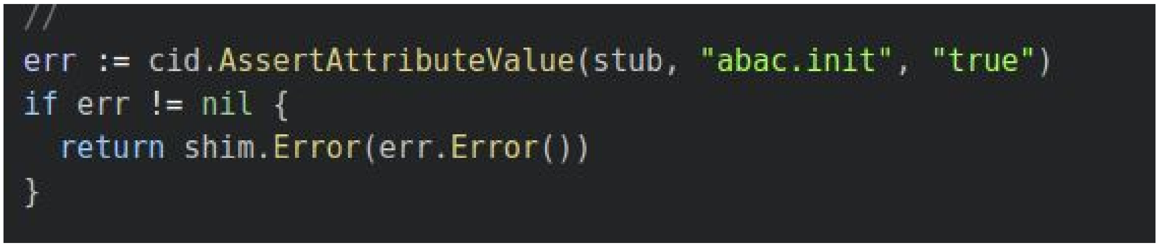
The scenario involves three organizations **org1** and **org2,** which are for the peer organizations; and **org0,** which is the orderer [92]. Six containers are started by the startup script: three root CAs, and three intermediate CAs. Each of the three organizations will be assigned two CA containers, one root and one intermediate. The root CAs execute the *fabric-ca-server* utility and create root CA certificates and make them available to entities on the network. The intermediate CAs enroll their respective root CAs and create their own certificates and make them available on the network. Next identities of the peers are registered and enrolled with the intermediate CA servers. When this happens, the administrator identity on the peer node Docker containers receives a certificate that provides permission for that administrator to execute chaincode on the network. The certificate of the administrator contains the attribute ***abacinit*** which is set to “true:ecert” [92]. This allows the admin user to execute chaincode. Figure 15 is a portion of the ***setup-fabric.sh*** script and shows a snippet of the code defining the attributes of the certificate. Application developers can set attributes of certificates for the application network.



```
t,abac.init=true:ecert"
```

Figure 15. Attribute abac.init is defined for certification. Source: [93].

The ***abac.init*** attribute represents the “Attribute-Based Access Control (ABAC)” feature associated with the use of CA servers. The ability to set attributes combined with the assertion function in the chaincode, pictured in Figure 16 allows control to be set regarding who can execute chaincode in the network.



```
//
err := cid.AssertAttributeValue(stub, "abac.init", "true")
if err != nil {
  return shim.Error(err.Error())
}
```

Figure 16. Golang code to assert access to chaincode. Source: [94].

The flow of logic in Figure 16 shows that if the selected attribute does not contain the proper value, the chaincode ends execution and reports an error.

After the CA operations have executed, the sample application continues with the scenario test case. The sample application starts with commands to bootstrap the CA containers [92]. All the containers are set up, the peers are joined to channels, and chaincode is installed, instantiated, and tested with queries. This information is relayed to the user through log messages through a terminal window. After the scenario completes, the last message points the user to the log file which contains additional details not shown in the terminal window.

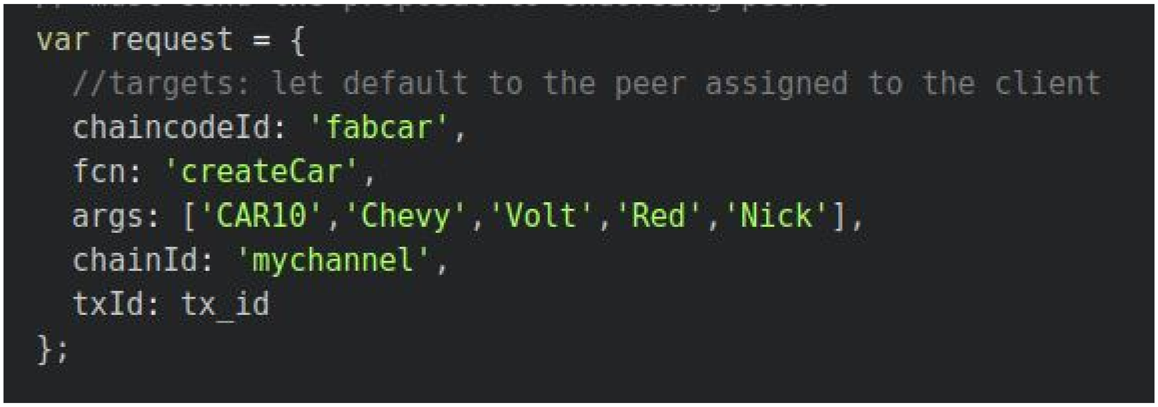
7. Fabcar

The ***Fabcar*** application executes a smaller scenario but demonstrates how an application is built and managed using Node.js and Hyperledger Fabrics existing code libraries. The scenario for the ***Fabcar*** example is a database of cars that the user queries for information or adds information to. The main application code is written in Node.js and the network is set up with a script that goes through the series of steps detailed in the ***Overview*** and ***First-Network*** sections. The documentation states that this application interacts with the API of Hyperledger Fabric to update and query the ledger [95]. Two files **query.js** and **invoke.js** manage queries of and updates to the ledger respectively. In both files, the request for a query, or an update is built in the JSON format.

Before the **query.js** and **invoke.js** files can be executed, users must enroll the administrator and register a non-admin user. This is done by executing the **enrollAdmin.js** and **registerUser.js** scripts. To show the messages that occur during CA enrollment and registration one can open another terminal and execute the command "`docker logs -f ca.example.com`" [95]. When this command is run, the waits for input and when **enrollAdmin.js** and **registerUser.js** are run, the terminal prints messages confirming enrollment and registration.

The script **query.js** returns all the entries on the ledger, but the query parameters can be edited to return different results. Unfortunately, the **invoke.js** script threw an error indicating that a function used in this script is no longer a part of the Node.js module provided by the developers. Researching this problem on forums yielded no results that worked to fix the issue. The proposal for updating the ledger with a new car in **invoke.js** is sent successfully, but functionality of this script ends here due to the error. This error is something that the developers should be able to fix in later versions of the platform. This error was also reported to the HLF development.

Figure 17 shows the request for updating the ledger in the ***Fabcar*** sample. The values in this JSON request define the function that will execute, the arguments for that function, and the channel or chain name. In this case the *createCar* function is run with five arguments that define the name, make, model, color, and owner of the car.



```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: 'createCar',
  args: ['CAR10','Chevy','Volt','Red','Nick'],
  chainId: 'mychannel',
  txId: tx_id
};
```

Figure 17. ***Fabcar*** ledger update request. Source: [96].

The CA operations for this example only include enrolling an administrator and non-administrator user. The two files **enrollAdmin.js**, and **registerUser.js** handle these operations. The admin user was registered with the CA, and the **enrollAdmin.js** file retrieves the certificate associated with that admin, which is then used to register and enroll the non-admin user in **registerUser.js** so they can interact with the ledger in this application [95].

B. CHANNEL BRANCHING AND MERGING

This section will propose an application design that uses the capabilities of a Hyperledger Fabric application in a way that not explicitly demonstrated in the documentation. Only the application design is described here; its implementation is future work. The discussion will cover the technical aspects of the chaincode function libraries provided by the developers for Golang and Node.js support.

Channels in Hyperledger Fabric are intended to be isolated from each other. The white paper for Hyperledger Fabric offers the idea that a blockchain can have confidentiality of execution of a smart contract so that only a subset of members on a channel can receive the results of that smart contract [33]. The documentation page for channels states that “no ledger data can pass from one channel to another,” which supports confidential communication between business partners [74]. Is there a way to merge information from one channel to another? Perhaps the capabilities of the HLF platform can be configured to make this work in theory.

1. Concept

The main idea is to archive transaction history data from one channel onto another HLF channel. In this proposed application, inventory data in key and value form, similar to JSON format, would be stored on the ledger for use in a short-term operation. The concept proposed here can work with either two channels, or one with channel and one database. In the scenario with two channels, one channel will contain all the keys and values associated with an initial inventory for the operation, and one channel will be given a subset of the keys and values from the first channel. In the second scenario, a database will be used to transfer data associated with the operation to an HLF channel in key-value form. The temporary channel that receives keys and values from the initial channel, or database in this application will be named the ***secondary channel*** and the channel or database that is used to initialize the ***secondary channel*** will be called ***primary channel***. The ***secondary channel*** will receive inventory keys and values, execute transactions, and then at some point, report back to the ***primary channel*** with its history of transactions involving those keys and values that have occurred up to that point in time. Figure 18 shows a notional diagram of how the application works.

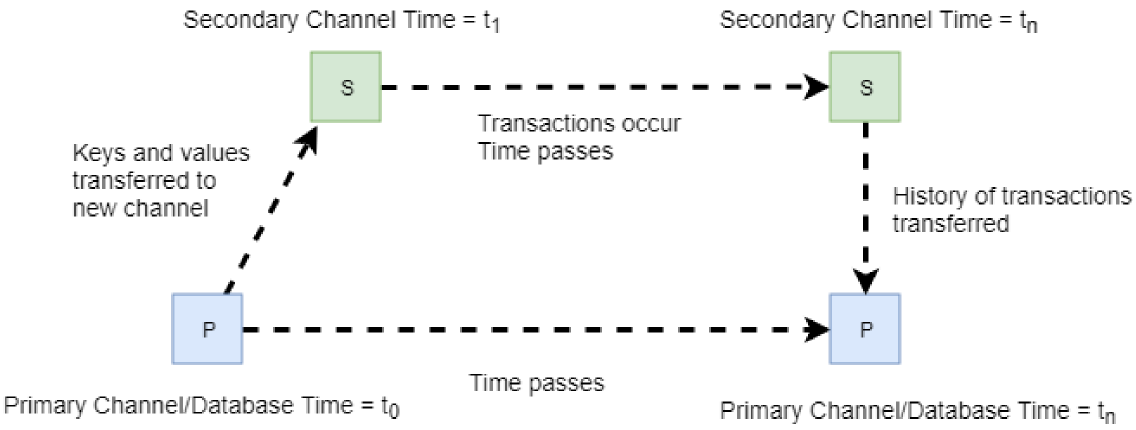

Secondary Channel Time = t1
Secondary Channel Time = tn
S
S
Transactions occur
Time passes
Keys and values
transferred to
new channel
History of transactions
transferred
P
P
Time passes
Primary Channel/Database Time = t0
Primary Channel/Database Time = tn

Figure 18. Channel merging notional diagram

In Figure 18 the ***primary channel*** starts at the time t_0 and transfers keys and values of some of its inventory to the ***secondary channel*** which starts at time t_1. Both channels continue with transactions and time passes. At time t_n the ***secondary channel*** must report the keys and values along with its transaction history. In this example, these keys and values represent data stored on the ledger. The ***secondary channel*** transfers the history of the transactions on the channel which includes the keys the ***primary channel*** initially transferred to it, the values associated with those keys, and any information on the actions that occurred between t_1 *and* t_n.

2. Necessary Components

Being able to obtain the history of keys and values on the ledger requires the ability to extract keys from the ledger of a channel. Multiple keys can be queried from a channel using the chaincode function *GetStateByRange*. The documentation for chaincode states that the *GetStateByRange* function takes two arguments, **startKey** and **endKey**, which returns an object that contains access to all the keys between the first key and the end key, including the first key and excluding the end key, in lexical order [97]. If both parameters are set to the empty string, there will be an unbounded range on all the keys that are returned. This is interpreted to mean that every key on the channel will be returned in this situation. Lexical order in this case means that the keys are sorted in the same way that words are sorted in a dictionary [98]. The return value of this function will be a

StateQueryInteratorInterface [97]. This object contains three documented functions: **HasNext**, which will return *true* if there is another key in the list, **Close**, which must be called when the object is no longer needed, and **Next**, which will return the next key and value in the list. The **Next** call returns a *KV* data structure that contains the key as a string, and the value as an array-like data structure of bytes [99].

Once keys are extracted from the ledger, they can be used with a combination of chaincode functions that can retrieve the transaction history associated with individual keys. The *GetHistoryForKey,* as documented, takes the key as a string argument and returns a *HistoryQueryIteratorInterface* object, which in turn has the functions **HasNext**, **Close**, and **Next** which work in the same way as the functions in the *StateQueryInteratorInterface* [97]. For this function the **Next** function returns a *KeyModification* object which contains the transaction identifier as a string, the value associated with that key again as an array-like structure, a timestamp variable, and a Boolean value named *IsDelete* which indicates if the key has been deleted at this point [99].

Once the history information is returned, the transaction identifier can be used with the Node.js SDK to retrieve further information about a transaction associated with a specific key. The Hyperledger Fabric SDK for Node.js provides the *queryTransaction* function, which takes a transaction identifier as a string argument and returns a *ProcessedTransaction* object [100]. This *ProcessedTransaction* object itself contains a *TransactionEnvelope* object [101]. The *TransactionEnvelope* object contains a *Transaction* object, which is an array of actions that took place in this transaction. This data structure would have to be parsed further to decide what information inside of it would be important to report back to the ***primary channel***.

3. Combining the Components

The first task is to write keys and values from the ***primary channel*** or database to the ***secondary channel***. For a channel, this can be accomplished with the *GetStateByRange* function implemented in chaincode for the ***primary channel***, allowing the application to collect a subset, or all the keys. For a database, keys and values can be queried and put into the JSON format. Then these keys and values can be parsed through the chaincode into basic string values and written to a location on a file system that can be read later. Once the values are written to disk, chaincode on the ***secondary channel*** can read them to instantiate the initial ledger state for that channel.

The next task is reporting the values associated with each key on the ***secondary channel*** at t_n. At this point, the chaincode on the ***secondary channel*** can execute the *GetStateByRange* function to collect keys for the next step. To collect all the keys from the ***secondary*** channel, the empty string should be provided to the *GetStateByRange* function so that every key on the channel is returned. In the chaincode associated with the ***secondary channel***, the next step is to iterate over the key list and use each key as an input argument to the *GetHistoryForKey* function to receive the *HistoryQueryIteratorInterface* object that contains the history associated with the key. For each key, the function will produce this object which should be parsed into string values that can be written to disk. In this case separate files for the history of each key could be created so that all data for a particular key is kept in. These history objects will contain a transaction identifier for each entry, which can be used in junction with the Node.js function *queryTransaction* to gain more information about specific transactions. A separate Node.js script would be necessary to access the key history files stored on disk and read transaction identifiers to use the *queryTransaction* function. The data returned from this would have to be parsed to decide what is important for reporting back to the ***primary channel***.

In the scenario containing two channels, once the information is collected from the ***secondary channel***, it can be written to the ledger through chaincode installed on peers in the ***primary channel***. The historical data would be written to the ***primary channel*** as new entries, where the key follows a naming convention for archived history, and the value is the collected historical data in JSON format.

In the scenario where ***primary*** is a database, once the information has been collected from the ***secondary channel***, it can be written to the database from disk. Chaincode on the ***secondary channel*** could directly write the historical data to the database, or an external script could be used to read the data from disk and enter it into the database. The historical data for each key would follow the format of the database.

After historical data is written to the ***primary channel*** or database, use of the ***secondary channel*** will be discontinued. In this model, a new channel will be created for each short-term operation and discarded once the operation is over.

This describes the basic concept for this HLF application. The capabilities of HLF allow for transaction data to be archived. Data could also be transferred between channels, something the platform was not intended to do, but can be made possible through a set of carefully designed steps. At this stage, custom development is necessary on the part of the party intending to use the HLF platform for this purpose and will be discussed in Chapter VII.

C. SUMMARY

This chapter discussed the sample applications available from the Hyperledger Fabric developers. The six sample applications covered a set of capabilities available to users and demonstrated the tools available for developing custom applications. Two of the sample applications, however, produced errors during their scenarios. These errors may be fixed in later versions of the HLF platform but their status was reported to the community of developers in case they were not aware of them. The other applications ran without error and demonstrated core concepts of the HLF platform.

Then an application concept was proposed for certain short-term operations. This design made use of available capabilities in the HLF platform to propose new behavior currently unavailable on the platform.

The next chapter will summarize the lessons learned from researching different blockchain platforms, as well as provide a perspective on the Hyperledger Fabric platform. The parts of the short-term operation application outlined here that require more work will be discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. FUTURE WORK AND CONCLUSIONS

This chapter will discuss the conclusions of reviewing blockchain platforms, selecting HLF from the set of BPs, and investigating the HLF platform. The results of setting up HLF on the testbed will also be discussed along with future work. This will include aspects of the proposed application discussed in Chapter VI that require more work.

A. FUTURE WORK

This section will discuss the areas of the proposed design from Chapter VI that need to be developed or developed further to make the it workable. The application may provide a way to transfer data between channels in HLF, something that the platform currently does not do. The application also will provide a way to archive the history of transactions on a channel. Certain chaincode and Node.js functions were identified in Chapter VI that could facilitate using existing HLF features to achieve the desired behavior.

1. Future Application Development

The proposed application from Chapter VI, channel history archiving, could work in theory given the capabilities and behavior allowed by the HLF platform. First, the application itself should be implemented in future works so that its feasibility can be assessed. In the ***Combining the Components*** section of Chapter VI, we noted that information that needs to be shared between channels should be written to disk. This step requires work to streamline and secure the process. An appropriate location for the transactional key and value information to be written to needs to be determined. The peers executing chaincode exist in their own containers and act like separate hosts on the application network. Because of this, the read and write operations for information to be transferred will require an appropriate protocol. The use of a database or storage server could support this operation. When supplied with a transaction identifier, the Node.js *queryTransaction* function returns an object that contains data about transactions. The returned object contains a lot of information, and more work is needed to determine much of the information returned is necessary to create an historical record of a transaction. If a channel executed many transactions on each key, the list returned from the

HistoryQueryIteratorInterface object could be long. An effective way to represent this data and the data returned from the *queryTransaction* function in human readable format must be developed. How data will be written back to the ***primary channel*** is a part of the application design that requires further analysis and development.

An application interface will be needed for this application. Ideally, a user should be able to open a graphical application window, or command line terminal and specify the ***primary channel***, the ***secondary channel*** to send keys and values to, and the keys associated with the inventory to transfer. The user should also be able to activate the transfer from ***primary*** to ***secondary*** and vice versa through the application interface.

2. Other Use for Hyperledger Fabric

Docker software allows for the creation of lightweight virtual machines. HLF can be deployed in these lightweight virtual machines instead of setting up distinct physical machines for a network. The labtainers framework provides a Docker-based environment for laboratory exercises for cybersecurity education [102]. The framework includes tools for developing new lab exercises [102]. Since HLF uses the Docker software, it should be possible to integrate these two entities in a future project. Exercises related to HLF operations could be designed for education in cybersecurity as well as the HLF platform. Students could then gain experience with blockchain platform techniques as well as cybersecurity techniques.

B. CONCLUSIONS

One of the lessons learned from reviewing blockchain platforms is that the information that developers make available about their platforms is not standardized. During the review process, we discovered the information we used for the metrics defined in Chapter III were not organized in the same way for every blockchain platform. Some websites provided links to community written electronic books, others provided a wiki page maintained by the developers, others described vague business solutions, whereas others provided huge documentation websites. Even though the platforms were available to download and use, they varied in maturity: some were started only a year ago, some had barely gotten past their first version 1.0 release and were still adding small features periodically, and, in one case, the company that had developed a platform pivoted to a new platform. Finding a blockchain platform that can support a variety of applications and their development was a challenge: a lot of platforms are based around tokens, or token exchange.

The testbed that we created was built on the Ubuntu 18.04 operating system hosted as a virtual machine on VirtualBox. The required software of Docker, Docker Compose, the Go language compiler, Node.js, and the HLF utilities, sample applications, and Docker images were installed. We were able to bring the HLF platform to a workable state so that the sample applications could be executed following the directions within the documentation for each sample application. Some of the sample applications could not be executed completely because of errors that arose. Specifically, the ***Fabcar***, and ***High-Throughput*** applications ran into errors with modules they were dependent on to complete their scenarios. Unfortunately, we were not able to overcome these issues in the time provided, even after searching forums and sample application files and code itself for solutions. This was the case because we were not sufficiently familiar with the backend code that the developers have designed. The functionality of the sample applications was demonstrated sufficiently by the documentation, however. The errors were reported to the development community for HLF so that these applications can be fixed.

Blockchain platforms can support the development of versatile applications. The cornerstone data structure, the blockchain, can be applied to different use cases, including distributed applications. HLF was chosen as the blockchain platform for the testbed because uses smart contracts and is not designed to require tokens or cryptocurrency [39]. Instead its developers made the platform as general as possible allowing for the design of a variety of distributed applications.

In closing, blockchain platforms that support smart contracts, PKI integration or similar capabilities have the possibility of being used by the DoD use. The DoD has already established a PKI that supports identification and authentication of users, as well as confidentiality and integrity of documents. This could be leveraged in HLF-based applications. The smart contract capability allows a blockchain platform to be tailored to the application needs of the user. However, care must be taken to select a blockchain platform that does not include expensive startup overhead costs. Instead, a platform that can be seamlessly integrated into current infrastructure is preferable.

APPENDIX A. BUILD ENVIRONMENT SETUP

This appendix adapts the following documents and tutorials to set up the build environment for the testbed network that will run Hyperledger Fabric in this study. This appendix serves as instructions on how to set up the test system involved in this study.

A. VIRUALIZING THE ENVIRONMENT

a. Required Software:

- VirtualBox software.
- Latest Ubuntu OS version (At the time of writing Ubuntu 18.04).

b. Set up by:

- Following Appendix A or B in the Labtainers instructional document available from the Naval Postgraduate School [102].
- The testbed environment was created on an Ubuntu 18.0.4 virtual machine for testing purposes.

B. INSTALL GIT

a. Needed:

- Git version control software.
- Follow the instructions on the Ubuntu guide for installing Git [103].
- Required for downloading Hyperledger Fabric samples and utilities, and the project repository.

C. INSTALL PREREQUISITES OF HYPERLEDGER FABRIC

1. Required Software

- Curl

- Docker and Docker Compose
- Golang

2. Installing Required Software

a. Detailed instructions on prerequisites available at:

- The prerequisites page in the Hyperledger Fabric documentation [104].

b. Install Curl by entering command:

- sudo apt install curl

c. Install docker

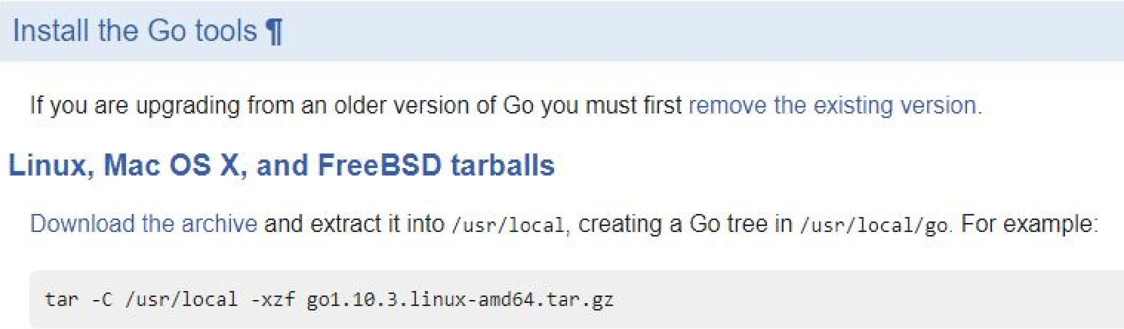
- follow the Docker documentation for Ubuntu [105].
- Additional instructions for installing Docker are in Appendix B.

d. Install Docker Compose

- follow the installation guide for Linux from the Docker website [106].
- Additional instructions for installing Docker Compose are in Appendix B

e. Install Golang

- Follow the link for Linux from the Golang download page [107]
- Note: Choose the Linux download link, 64-bit processor required.
- Follow the installation instructions for Linux after the download link is clicked. Installation instructions are pictured in Figure 19 and Figure 20.



Install the Go tools ¶

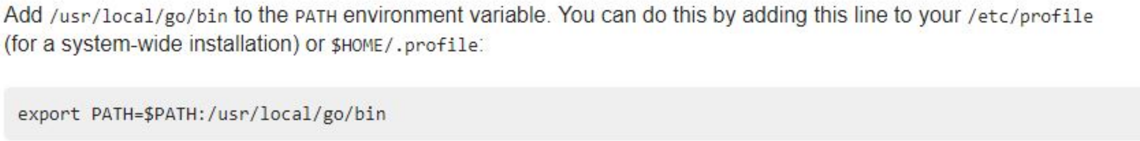
If you are upgrading from an older version of Go you must first remove the existing version.

Linux, Mac OS X, and FreeBSD tarballs

Download the archive and extract it into /usr/local, creating a Go tree in /usr/local/go. For example:

```
tar -C /usr/local -xzf go1.10.3.linux-amd64.tar.gz
```

Figure 19. Instructions for extracting Golang files. Source: [108].



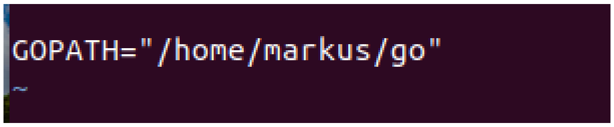
Add /usr/local/go/bin to the PATH environment variable. You can do this by adding this line to your /etc/profile (for a system-wide installation) or $HOME/.profile:

```
export PATH=$PATH:/usr/local/go/bin
```

Figure 20. Directions for adding Golang binary directory to PATH variable. Source: [108].

f. Adding environment variables

- Will need to create a directory to hold chaincode for Hyperledger-Fabric
- Can be on any location on the disk, must be named 'go' for example: `$HOME/go`
- Directory must be put into environment variables as `$GOPATH`.
- Edit `/etc/environment` and add the line `GOPATH = $HOME/go`
- Replace '$HOME' with chosen path as below if needed. Example GOPATH variable in Figure 21.



```
GOPATH="/home/markus/go"
~
```

Figure 21. Format of the environment variable in `/etc/environment` file

g. Install Node.js on Ubuntu

- Follow the instructions from the Digital Ocean Node.js installation guide for Ubuntu 18.04 [109].

- Note: The instructions should work on later versions of Ubuntu as well.

D. HYPERLEDGER FABRIC BINARIES AND SAMPLES

1. Binaries

a. Install binaries

- Follow the Hyperledger Fabric documentation starting at the binaries section of the samples page [110].

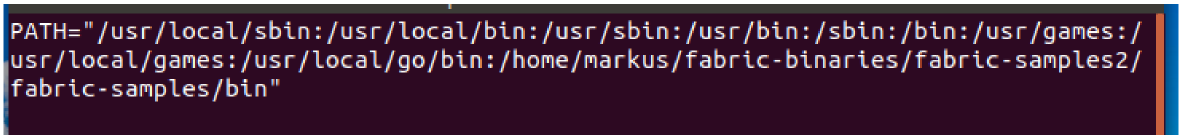
- Updated versions of documentation might exist. If so, follow the updated directions.

b. Note: modify the curl command to read:

- sudo curl -sSL <URL> | sudo bash -s <version number>

c. Modify PATH variable

- Add the path to the bin directory for the binaries to the `/etc/environment` file. Figure 22 shows an example of a modified PATH variable to include the binary directory path for the Hyperledger Fabric samples.



```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
usr/local/games:/usr/local/go/bin:/home/markus/fabric-binaries/fabric-samples2/
fabric-samples/bin"
```

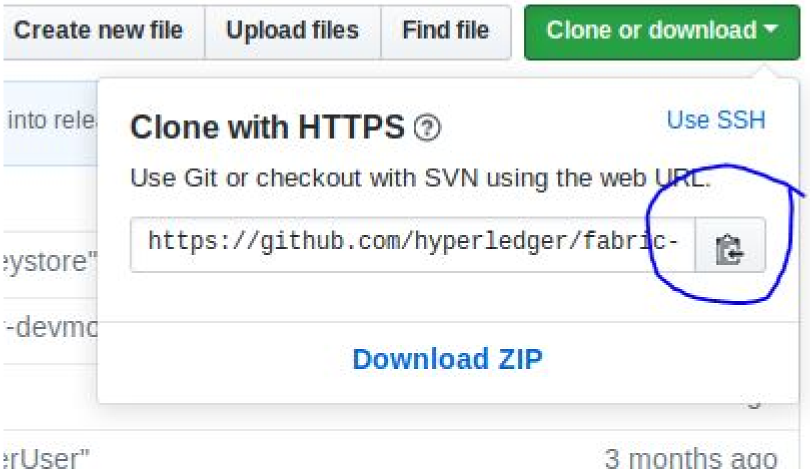
Contains the absolute path to the `bin` directory for the Hyperledger Fabric binaries

Figure 22. The path variable modified in the `/etc/environment` file

2. Samples

The samples are scenarios and use cases in Hyperledger fabric, follow the instructions below to install them:

- Instructions available on the Hyperledger Fabric samples page of the documentation [110].
- Note: the documentation website provides directions for cloning the git repository
- Alternatively, the repository can be downloaded directly from the GitHub page for the Hyperledger Fabric samples [111].
- Note: New versions may be released after time of writing, but the procedure to copy the repository will be the same.
- Select the option to copy the repository so it can be cloned in a directory of choice
- Figure 23 through Figure 25 show the steps to take for cloning the repository and verifying that the samples are in the directory.



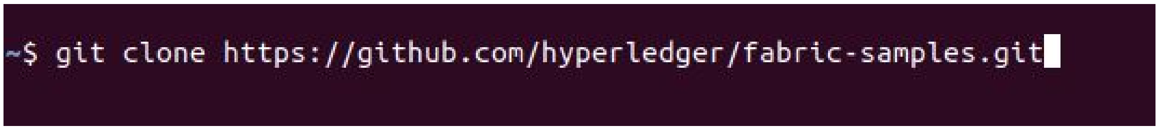
Clone with HTTPS

Use Git or checkout with SVN using the web URL.

https://github.com/hyperledger/fabric-

Click the clipboard icon in the circled area to copy the repository link to the clipboard.

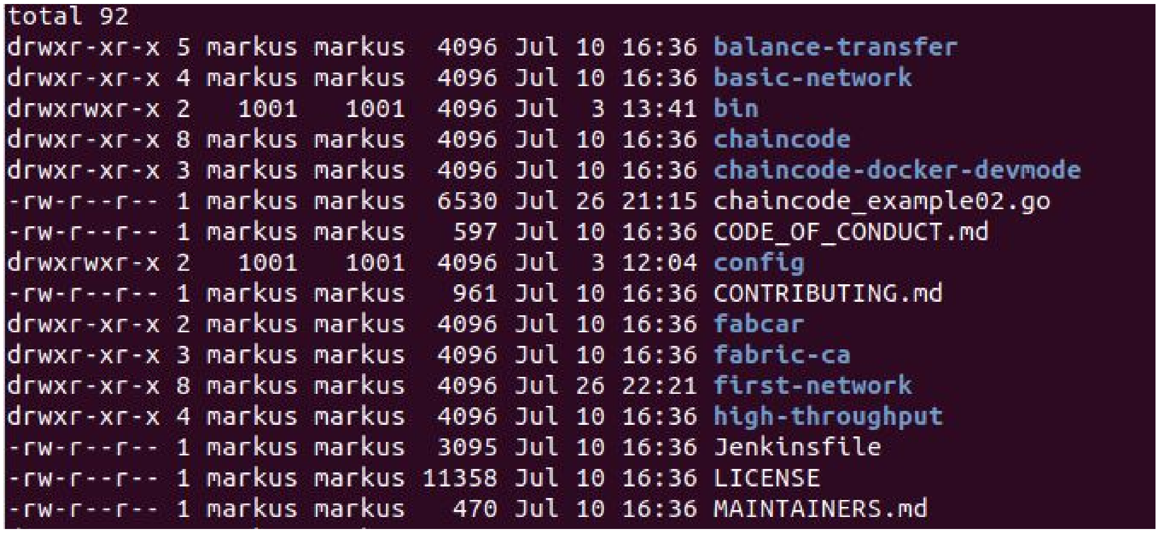
Figure 23. Dropdown menu on GitHub repository. Source: [111].



```
~$ git clone https://github.com/hyperledger/fabric-samples.git
```

The hyperlink was pasted from the clipboard after being copied from GitHub.

Figure 24. Git clone command for fabric-samples repository



```
total 92
drwxr-xr-x 5 markus markus  4096 Jul 10 16:36 balance-transfer
drwxr-xr-x 4 markus markus  4096 Jul 10 16:36 basic-network
drwxrwxr-x 2   1001   1001  4096 Jul  3 13:41 bin
drwxr-xr-x 8 markus markus  4096 Jul 10 16:36 chaincode
drwxr-xr-x 3 markus markus  4096 Jul 10 16:36 chaincode-docker-devmode
-rw-r--r-- 1 markus markus  6530 Jul 26 21:15 chaincode_example02.go
-rw-r--r-- 1 markus markus   597 Jul 10 16:36 CODE_OF_CONDUCT.md
drwxrwxr-x 2   1001   1001  4096 Jul  3 12:04 config
-rw-r--r-- 1 markus markus   961 Jul 10 16:36 CONTRIBUTING.md
drwxr-xr-x 2 markus markus  4096 Jul 10 16:36 fabcar
drwxr-xr-x 3 markus markus  4096 Jul 10 16:36 fabric-ca
drwxr-xr-x 8 markus markus  4096 Jul 26 22:21 first-network
drwxr-xr-x 4 markus markus  4096 Jul 10 16:36 high-throughput
-rw-r--r-- 1 markus markus  3095 Jul 10 16:36 Jenkinsfile
-rw-r--r-- 1 markus markus 11358 Jul 10 16:36 LICENSE
-rw-r--r-- 1 markus markus   470 Jul 10 16:36 MAINTAINERS.md
```

Figure 25. Snippet of directory listing from fabric-samples directory

- After the link is copied navigate to any desired directory and execute the command: git clone <repository link>

E. TESTING THE SYSTEM IS INSTALLATION

Run through one of the scenarios provided by the Hyperledger Fabric documentation to make sure that everything is installed correctly.

- A tutorial for verifying functionality can be found on the “building your first network” page in the Hyperledger Fabric documentation [75].

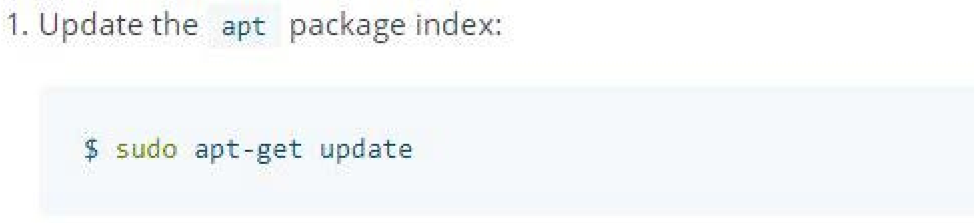
- Note: The tutorial only needs to be followed up to the “Bring Down the Network” section.

APPENDIX B. DOCKER AND DOCKER COMPOSE SETUP

This appendix will provide step by step instructions for installing Docker and Docker Compose. Screenshots are provided for archival purposes. The instructions captured from the website could become obsolete with newer updates to the software, but at the time of writing they are the current instructions.

A. DOCKER INSTALLATION INSTRUCTIONS

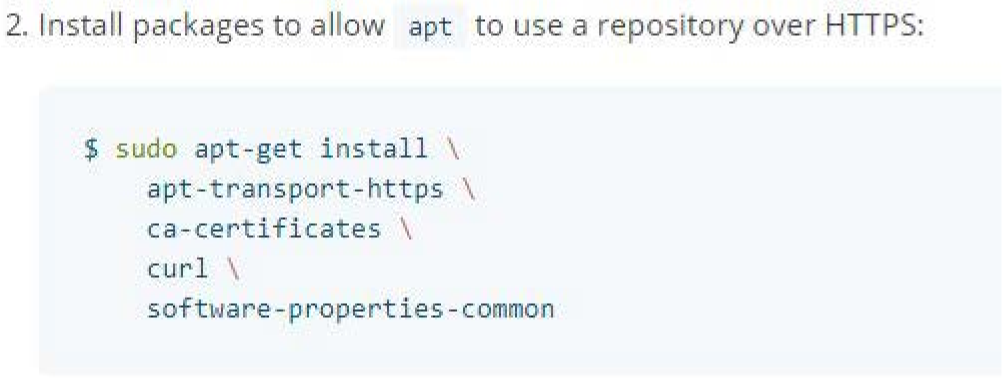
Figure 26 to Figure 33 show the instructions for installing Docker on an Ubuntu Linux system. At the time of writing the current Ubuntu version is 18.04.1, which is compatible with these instructions. These instructions should work on later versions of Ubuntu as well. If they do not, it is recommended to find an updated version of these instructions at the Docker website.



1. Update the `apt` package index:

```
$ sudo apt-get update
```

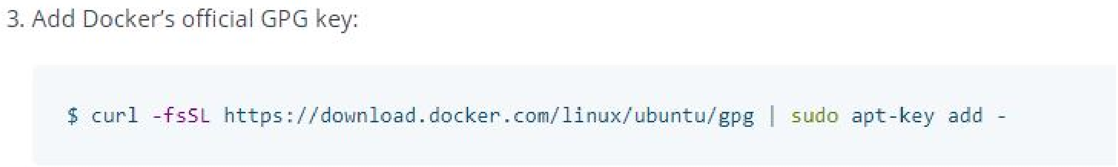
Figure 26. Part 1—Setting up the repository. Source: [105].



2. Install packages to allow `apt` to use a repository over HTTPS:

```
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

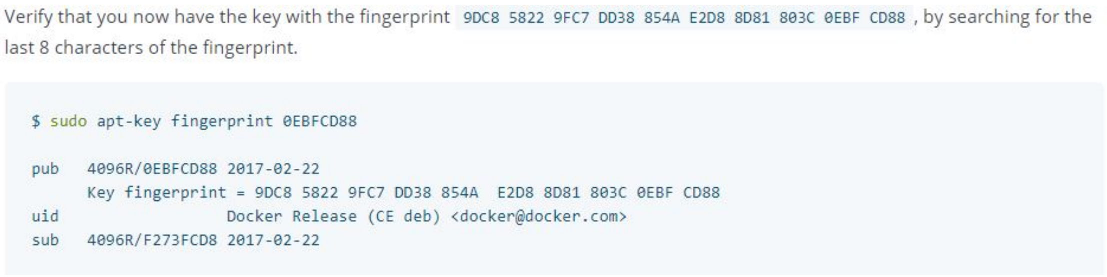
Figure 27. Part 2—Installing required software. Source: [105].



3. Add Docker’s official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Figure 28. Part 3—Download Docker’s GPG key. Source: [105].

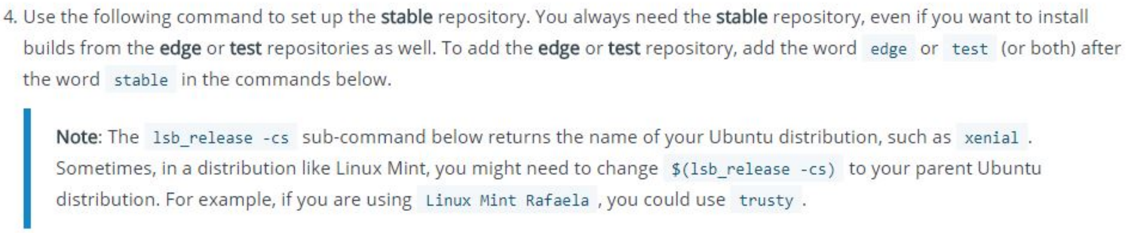


Verify that you now have the key with the fingerprint `9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88`, by searching for the last 8 characters of the fingerprint.

```
$ sudo apt-key fingerprint 0EBFCD88

pub   4096R/0EBFCD88 2017-02-22
      Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid                  Docker Release (CE deb) <docker@docker.com>
sub   4096R/F273FCD8 2017-02-22
```

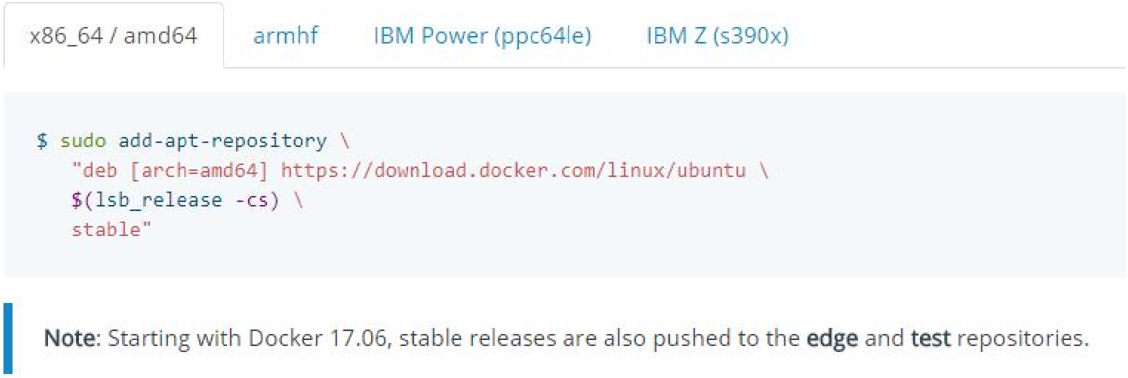
Figure 29. Part 3b—Verify key. Source: [105].



4. Use the following command to set up the **stable** repository. You always need the **stable** repository, even if you want to install builds from the **edge** or **test** repositories as well. To add the **edge** or **test** repository, add the word `edge` or `test` (or both) after the word `stable` in the commands below.

> **Note**: The `lsb_release -cs` sub-command below returns the name of your Ubuntu distribution, such as `xenial`. Sometimes, in a distribution like Linux Mint, you might need to change `$(lsb_release -cs)` to your parent Ubuntu distribution. For example, if you are using `Linux Mint Rafaela`, you could use `trusty`.

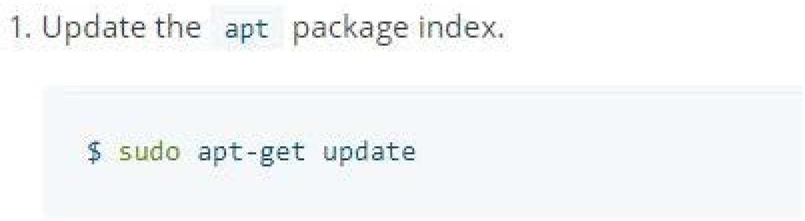
Figure 30. Part 4a—Repository update instructions. Source: [105].



```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

Note: Starting with Docker 17.06, stable releases are also pushed to the **edge** and **test** repositories.

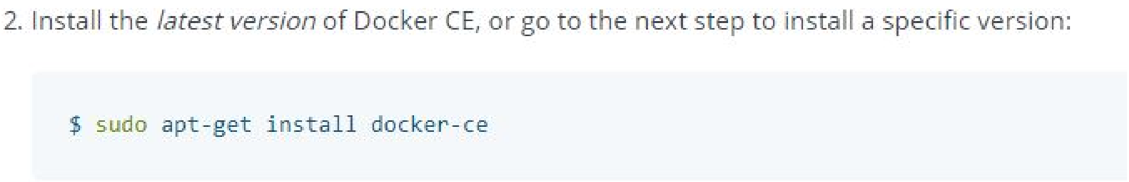
Figure 31. Part 4b—Linux command to setup stable repository. Source: [105].



1. Update the `apt` package index.

```
$ sudo apt-get update
```

Figure 32. Part 5—Update command. Source: [105].



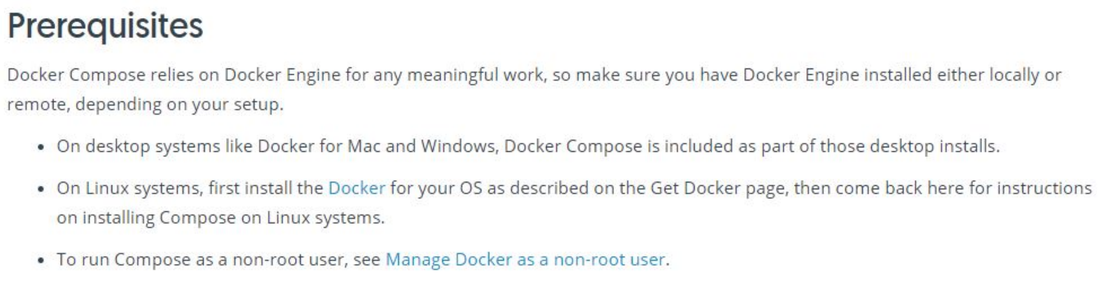
2. Install the *latest version* of Docker CE, or go to the next step to install a specific version:

```
$ sudo apt-get install docker-ce
```

Figure 33. Part 6—Docker installation command. Source: [105].

B. DOCKER COMPOSE INSTALLATION INSTRUCTIONS

Figure 34 to Figure 37 show the instructions for installing Docker Compose and assume that Docker is already installed on the system. Again, these instructions are current at the time of writing, but should they become obsolete it will be necessary to find the updated instructions from the docker website.

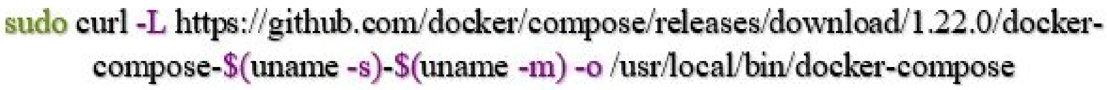


Prerequisites

Docker Compose relies on Docker Engine for any meaningful work, so make sure you have Docker Engine installed either locally or remote, depending on your setup.

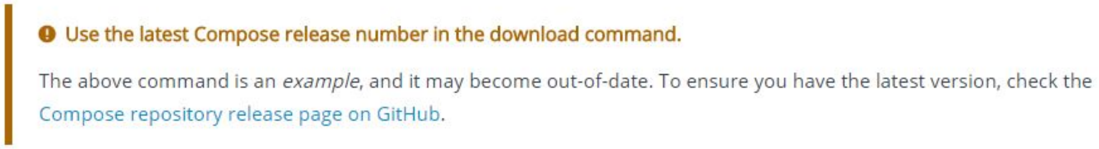
- On desktop systems like Docker for Mac and Windows, Docker Compose is included as part of those desktop installs.
- On Linux systems, first install the Docker for your OS as described on the Get Docker page, then come back here for instructions on installing Compose on Linux systems.
- To run Compose as a non-root user, see Manage Docker as a non-root user.

Figure 34. Prerequisites for Docker Compose installation. Source: [106].



```
sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose
```

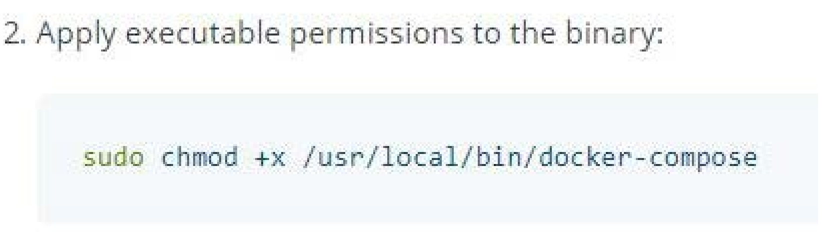
Figure 35. Part 1a—Docker Compose download command using curl. Source: [106].



❶ **Use the latest Compose release number in the download command.**

The above command is an *example*, and it may become out-of-date. To ensure you have the latest version, check the Compose repository release page on GitHub.

Figure 36. Part 1b—Notice on Docker Compose version number. Source: [106].



2. Apply executable permissions to the binary:

```
sudo chmod +x /usr/local/bin/docker-compose
```

Figure 37. Part 2—Changing permissions for Docker Compose binary. Source: [106].

LIST OF REFERENCES

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Bitcoin Project, 31 October 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] M. Swan, *Blockchain*. Sebastopol, CA, USA: O’Reilly Media, Inc, 2015.

[3] I. Bashir, *Mastering Blockchain*, 2nd ed. Birmingham, UK: Packt Publishing, 2018

[4] S. Raval, *Decentralized Applications*. Sebastopol, CA, USA: O’Reilly Media, Inc, 2016.

[5] N. Prusty, *Building Blockchain Projects*. Birmingham, UK: Packt Publishing, 2017

[6] “Consensus mechanisms explained,” 3iQ Corp, 5 April 2018. [Online]. Available: https://3iq.ca/3iq-research-group/consensus-mechanisms/

[7] “Blockchain basics,” 3iQ Corp, 29 December 2017. [Online]. Available: https://3iq.ca/3iq-research-group/blockchain-basics-2/

[8] L. Schor, “On zero-knowledge proofs in blockchains,” *Medium*, 23 March 2018. [Online]. Available: https://medium.com/@argongroup/on-zero-knowledge-proofs-in-blockchains-14c48cfd1dd1

[9] L. Lamport, R. Shostak and M. Pease, “The Byzantine generals problem,” *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 3824–01, 1982.

[10] G. Konstantopoulos, “Understanding blockchain fundamentals, part1: Byzantine fault tolerance,” *Medium*, 30 November 2017. [Online]. Available: https://medium.com/loom-network/understanding-blockchain-fundamentals-part-1-byzantine-fault-tolerance-245f46fe8419

[11] M. Castro and B. Liskov, “Practical Byzantine fault tolerance,” in *Proceedings of theThird Symposium on Operating Systems Design and Implementation*, New Orleans, LA, USA, 1999.

[12] S. Chacon and B. Straub, *Pro Git*, 2nd ed. New York, NY, USA: Apress, 2014.

[13] E. Sink, *Version Control by Example*,. Champaign, IL, USA: Pyrenean Gold Press, 2011.

[14] “Blockchain forks,” 3iQ Corp, 18 February 2018. [Online]. Available: https://3iq.ca/3iq-research-group/blockchain-forks/

[15] R. Modi, *Solidity Programming Essentials*. Birmingham, UK: Packt Publishing, 2018.

[16] A. Narayanan and J. Clark, “Bitcoin’s academic pedigree,” *Communications of the ACM*, vol. 60, no. 12, pp. 364–5, December 2017.

[17] A. Antonopolous, *Mastering Bitcoin*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media Inc, 2017.

[18] C. Dwork and M. Naor, “Pricing via processing or combatting junk mail,” in *Advances in Cryptology -- Crypto ‘92*, 1992. pp 139 -147. [Online]. Available: https://dl.acm.org/citation.cfm?id=705669

[19] V. Beal, “Public-key encryption,” QuinStreet Inc, accessed 22 July 2018. [Online]. Available: https://www.webopedia.com/TERM/ P/public_key_cryptography.html

[20] W. Mougayar and V. Buterin, *The Business Blockchain*. Hoboken, NJ, USA: John Wiley & Sons, 2016.

[21] “Chain of custody,” Duke Law, accessed 3 July 2018. [Online]. Available: http://www.edrm.net/glossary/chain-of-custody/

[22] M. Meyers and S. Jernigan, *Mike Meyers’ CompTIA Security+ Certification Guide*, 2nd ed. New York City, NY, USA: McGraw Hill, 2017.

[23] P. Fairley, “Feeding the blockchain beast if Bitcoin ever does go mainstream, the electricity needed to sustain it will be enormous,” *IEEE Spectrum*, vol. 54, no. 10, pp. 36–59, Sep. 2017. [Online]. doi: 10.1109/MSPEC.2017.8048837

[24] “Red Hat support,” Red Hat, Inc., accessed 21 June 2018. [Online]. Available: https://www.redhat.com/en/services/support

[25] Mahnke, “LTS,” Ubuntu, 17 March 2017. [Online]. Available: https://wiki.ubuntu.com/LTS

[26] “Red Hat subscription model FAQ,” Red Hat, Inc., accessed 1 August 2018. [Online]. Available: https://www.redhat.com/en/about/subscription-model-faq

[27] “Frequently asked questions about the GNU licenses,” Free Software Foundation, Inc., 17 June 2018. [Online]. Available: https://www.gnu.org/licenses/gpl-faq.html#GPLRequireSourcePostedPublic

[28] “Enterprise software licensing: New options, new headaches,” *IDG Communications Inc.*, 16 March 2010. [Online]. Available: https://www.cio.com/ article/2419740/enterprise-software/enterprise-software-licensing--new-options--new-headaches.html

[29] P. Jayachandran, “The difference between public and private blockchain,” IBM, 31 May 2017. [Online]. Available: https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/

[30] A. Kadiyala, “Nuances between permissionless and permissioned blockchains,” *Medium*, a 4 August 2018. [Online]. Available: https://medium.com/@akadiyala/nuances-between-permissionless-and-permissioned-blockchains-f5b566f5d483

[31] M. Fowler and R. Parsons, *Domain Specific Languages*. Boston, MA, USA: Addison-Wesley, 2014. [Online]. Available: https://martinfowler.com/books/dsl.html

[32] "Definition of: General-purpose language," *PCMag Digital Group*, accessed 31 August 2018. [Online]. Available: https://www.pcmag.com/encyclopedia/term/43726/general-purpose-language

[33] E. Androulaki, et al., “Hyperledger Fabric: A distributed operating system for permissioned blockchains,” in *EuroSys*, article no: 30, April 2018. [Online]. doi: 10.1145/3190508.3190538

[34] T. Kosar and e. al, "Comparing general-purpose and domain-specific anguages: An empirical study," *ComSIS*, vol. 7, no. 2, pp. 247-264, 2010. [Online]. doi: 10.2298/CSIS1002247K

[35] G. Tomassetti, "Discovering the arcane world of esoteric programming languages," *Tomassetti*, accessed 31 August 2018. [Online]. Available: https://tomassetti.me/discovering-arcane-world-esoteric-programming-languages/

[36] “TIOBE index,” TIOBE Software BV, accessed 4 August 2018. [Online]. Available: https://www.tiobe.com/tiobe-index/

[37] “API definition,” Sharpend Productions, 20 June 2016. [Online]. Available: https://techterms.com/definition/api

[38] "Awesome-blockchain," Github, accessed 21 April 2018. [Online]. Available: https://github.com/imbaniac/awesome-blockchain

[39] “Chaincode Tutorials,” Hyperledger, accessed 10 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/chaincode.html

[40] “Smart contracts/EVM FAQ,” Counterparty, accessed 4 August 2018. [Online]. Available: https://counterparty.io/docs/faq-smartcontracts/#why-would-an-ethereum-developer-develop-on-counterparty

[41] “Hyperledger Burrow,” The Linux Foundation, accessed 4 August 2018. [Online]. Available: https://www.hyperledger.org/projects/hyperledger-burrow

[42] C. Kihlman, "Monax is pivoting. Here's why!," Monax, 26 April 2018. [Online]. Available: https://monax.io/blog/2018/04/26/monax-is-pivoting.-heres-why/

[43] "What is Expanse," Expanse.tech, 9 March 2018. [Online]. Available: https://expanse.tech/docs/introduction/what-is-expanse/

[44] “Expanse,” Expanse.tech, accessed 4 August 2018. [Online]. Available: https://expanse.tech/

[45] “Waves node rest api,” Waves, 19 July 2018. [Online]. Available: https://docs.wavesplatform.com/en/development-and-api/waves-node-rest-api.html

[46] "Assets," Waves, 19 July 2018. [Online]. Available: https://docs.wavesplatform.com/en/platform-features/assets-custom-tokens.html

[47] “Asset exchange,” NXT, accessed 30 July 2018. [Online]. Available: https://nxtplatform.org/what-is-nxt/asset-exchange/

[48] “Monetary system,” NXT, accessed 30 July 2018. [Online]. Available: https://nxtplatform.org/what-is-nxt/monetary-system/

[49] G. Greenspan, “Beware the impossibe smart contract,” Multichain, 12 April 2016. [Online]. Available: https://www.multichain.com/blog/2016/04/beware-impossible-smart-contract/

[50] nopara73, “Programming the blockchain in C# community edition,” accessed 22 June 2018. [Online]. Available: https://programmingblockchain.gitbooks.io/programmingblockchain/content/introduction/why_c.html

[51] I. McCain, “Stratis smart contracts in C# alpha release,” Stratis, 16 May 2018. [Online]. Available: https://stratisplatform.com/2018/05/16/stratis-smart-contracts-in-csharp/

[52] "DCore technical description," Decent, accessed 28 August 2018. [Online]. Available: https://docs.decent.ch/DCoreTechDesc/index.html#consensus_algorithm

[53] “Frequently asked questions,” Factom, accessed 4 August 2018. [Online]. Available: https://www.factom.com/about/faqs

[54] "Core building blocks," Brainbot Technologies, accessed 28 August 2018. [Online]. Available: http://www.brainbot.com/projects.html

[55] "Hydrachain/hydrachain," Brainbot Technologies, accessed 30 July 2018. [Online]. Available: https://github.com/HydraChain/hydrachain

[56] “What is Docker?,” Open Source, accessed 5 August 2018. [Online]. Available: https://opensource.com/resources/what-docker

[57] “Docker overview,” Docker Inc., accessed 22 June 2018. [Online]. Available: https://docs.docker.com/engine/docker-overview/

[58] “Overview of Docker Compose,” Docker Inc., accessed 22 June 2018. [Online]. Available: https://docs.docker.com/compose/overview/

[59] “Apache CouchDB,” Apache Software Foundation, accessed 13 July 2018. [Online]. Available: http://couchdb.apache.org/

[60] “CouchDB as the state database,” Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/couchdb_as_state_database.html

[61] E. T. Bray, "The JavaScript object notation (JSON) data interchange format," Internet Engineering Task Force, December 2017. [Online]. Available: https://tools.ietf.org/html/rfc8259

[62] R. Pike, "Go at Google: Language design in the service of software engineering," 25 October 2012. [Online]. Available: https://talks.golang.org/2012/splash.article#TOC_1

[63] “About Nodejs,” Node.js Foundation, accessed 10 July 2018. [Online]. Available: https://nodejs.org/en/about/

[64] J. Snellinckx, “How to start writing your Hyperledger Fabric Nodejs chaincode,” *Medium*, accessed 5 August 2018. [Online]. Available: https://medium.com/wearetheledger/how-to-start-writing-your-hyperledger-fabric-nodejs-chaincode-4052393933ab

[65] “The Go programming language documentation,” Golang, accessed 5 August 2018. [Online]. Available: https://golang.org/doc/

[66] C. Tartamella, “Node.Js: Just what is it?,” Anexinet, 7 November 2016. [Online]. Available: https://www.anexinet.com/blog/node-js-just/

[67] J. Haas, “The difference between compiled and interpreted languages,” *Lifewire*, 16 April 2018. [Online]. Available: https://www.lifewire.com/compiled-language-2184210

[68] “Chaincode for developers,” Hyperledger, accessed 10 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/chaincode4ade.html

[69] “Chaincode for operators,” Hyperledger, accessed 10 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/chaincode4noah.html

[70] “Architecture explained,” Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/arch-deep-dive.html

[71] M. Paul, “Hyperledger -- Chapter 10| blockchain application on Hyperledger Fabric,” *Medium*, accessed 6 August 2018. [Online]. Available: https://medium.com/swlh/hyperledger-chapter-10-blockchain-application-on-hyperledger-fabric-2e34f3f430b

[72] “Glossary,” Hyperledger, accessed 12 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/glossary.html

[73] “Membership,” Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/membership/membership.html

[74] “Channels,” Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/channels.html

[75] "Building your first network," Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/build_network.html

[76] “Hyperledger Fabric model,” Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/fabric_model.html

[77] “Transaction flow,” Hyperledger, accessed 13 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/txflow.html

[78] "Hyperledger Fabric sdks," Hyperledger, accessed 11 September 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/fabric-sdks.html

[79] L. Kolisko, "Hyperledger fabric-sdk-java basics tutorial," *Medium*, 20 February 2018. [Online]. Available: https://medium.com/@lkolisko/hyperledger-fabric-sdk-java-basics-tutorial-a67b2b898410

[80] “Cryptogen commands,” Hyperledger, accessed 22 June 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/cryptogen-commands.html

[81] “Configtxgen,” Hyperledger, accessed 22 June 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/configtxgen.html

[82] “Peer,” Hyperledger, accessed 12 July 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/commands/peercommand.html

[83] “Fabric-ca commands,” Hyperledger, accessed 9 August 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/fabric-ca-commands.html

[84] “YAML about page,” YAML, accessed 5 July 2018. [Online]. Available: http://yaml.org/about.html

[85] “YAML syntax,” Red Hat, Inc., accessed 5 July 2018. [Online]. Available: https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html

[86] "Basic network config," Hyperledger Fabric, accessed 14 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/release-1.2/basic-network/README.md

[87] "Fabricamples/basic-network/start.sh," Hyperledger, accessed 14 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/release-1.2/basic-network/start.sh

[88] "HTTP request methods," W3Schools, accessed 27 August 2018. [Online]. Available: https://www.w3schools.com/tags/ref_httpmethods.asp

[89] "Balance transfer," Hyperledger, accessed 14 August 2018.[Online]. Available: https://github.com/hyperledger/fabric-samples/blob/release-1.2/balance-transfer/README.md

[90] "High-throughput network," Hyperledger, accessed 25 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/release-1.2/high-throughput/README.md

[91] M. Rouse, "Race condition," TechTarget, accessed 25 August 2018. [Online]. Available: https://searchstorage.techtarget.com/definition/race-condition

[92] "Fabric-samples/fabric-ca," Hyperledger, accessed 23 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/tree/release-1.2/fabric-ca

[93] "Fabric-samples/fabric-ca/scripts/setup-fabric.sh," Hyperledger, accessed 25 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/release-1.2/fabric-ca/scripts/setup-fabric.sh

[94] "Fabric-samples/chaincode/abac/go," Hyperledger, accessed 25 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/release-1.2/chaincode/abac/go/abac.go

[95] "Writing your first application," Hyperledger, accessed 25 August 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.2/ write_first_app.html

[96] "Fabric-samples/fabcar," Hyperleger, accessed 25 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples/tree/release-1.2/fabcar

[97] "Fabric/core/chaincode/shim/interfaces.go," Hyperledger, accessed 12 July August. [Online]. Available: https://github.com/hyperledger/fabric/blob/release-1.2/core/chaincode/shim/interfaces.go

[98] "Lexical order," William Collins Sons & Co. Ltd., accessed 26 August 2018. [Online]. Available: https://www.dictionary.com/browse/lexical-order

[99] “Package queryresult," Hyperledger, accessed 25 July 2018. [Online]. Available: https://godoc.org/github.com/hyperledger/fabric/protos/ledger/queryresult

[100] “Class: Channel," Hyperledger, accessed 26 August 2018. [Online]. Available: https://fabric-sdk-node.github.io/Channel.html#queryTransaction

[101] "Global," Hyperledger, accessed 26 August 2018. [Online]. Available: https://fabric-sdk-node.github.io/global.html

[102] "Labtainer student guide," Naval Postgraduate School, 16 August 2018. [Online]. Available: https://my.nps.edu/documents/107523844/109121513/labtainer-student.pdf/a6c49134-4bd4-457f-bb88-f5cb79b34fd0

[103] “Git,” Ubuntu, accessed 8 June 2018. [Online]. Available: https://help.ubuntu.com/lts/serverguide/git.html.en

[104] “Prerequisites,” Hyperledger, accessed 9 July 2018. [Online]. Available: http://hyperledger-fabric.readthedocs.io/en/release-1.2/prereqs.html

[105] “Get Docker CE for Ubuntu,” Docker, accessed 7 August 2018. [Online]. Available: https://docs.docker.com/install/linux/docker-ce/ubuntu/

[106] “Install Docker Compose,” Docker, accessed 7 August 2018. [Online]. Available: https://docs.docker.com/compose/install/

[107] “Downloads,” Golang, accessed 7 June 2018. [Online]. Available: https://golang.org/dl/

[108] "Getting started," Golang, accessed 7 August 2018. [Online]. Available: https://golang.org/doc/install?download=go1.10.3.linux-amd64.tar.gz

[109] K. Juell, "How to install Node.js on Ubuntu 18.04," Digital Ocean, 27 April 2018. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-18-04

[110] "Hyperledger Fabric samples," Hyperledger, accessed 20 May 2018. [Online]. Available: http://hyperledger-fabric.readthedocs.io/en/release-1.1/samples.html

[111] "Hyperledger/fabric-samples," Hyperledger, accessed 7 August 2018. [Online]. Available: https://github.com/hyperledger/fabric-samples

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California